SimFlex & ProtoFlex

- Fast and Accurate Full-System Simulation
- Tutorial & Hands-on Session

- Before we begin, please open:
  - http://www.ece.cmu.edu/~protoflex/
  - Ask Michael Papamichael for login information

SimFlex & ProtoFlex

Michael Ferdman

Computer Architecture Lab at Carnegie Mellon

December 4, 2010

Overview

- Simics full-system simulation environment (mini)
  - Hands-on: Using Simics to bootstrap the simulation
- SimFlex: full-system simulation infrastructure
  - Lecture: Flexus trace and cycle-accurate simulators
    - Hands-on: Simulating with Flexus
  - Lecture: SMARTS simulation methodology
    - Hands-on: Accelerating Flexus simulation with SMARTS
- ProtoFlex: FPGA architectural exploration
  - Lecture: ProtoFlex concepts and implementation
    - Hands-on: Using ProtoFlex for architectural exploration

Simics Full-System Simulator

Michael Ferdman

Computer Architecture Lab at Carnegie Mellon

December 4, 2010

Why We Use Simics

- Problem: Full-system simulation is hard
  - I/O device handling is tricky
  - Some instructions (especially privileged ones) are hard
  - Lots of infrastructure (disk formats, CLI, checkpointing, etc…)
- Solution: Leverage someone else's work (Simics)
  - Implements network, disk, video, and all other I/O devices
  - Faithfully models all gory CPU details to boot real OSes
  - Well-designed CLI, full-system checkpoints, scripting API, etc…

Simics Basics

- Configuration file defines system components
  - CPUs, motherboard, memory, disks, video card, …
- CLI provides interface to simulation
  - `read-configuration system-config.simics`
    - Read configuration file
  - `write-configuration system-config.simics`
    - Write out complete system checkpoint
  - `run 100`
    - Execute 100 instructions per CPU

Simics Hands-on

- Booting system
- Logging into simulated system
- Interrupting execution
  - Examining the simulated system's registers (`pregs`)
- Taking system checkpoints
- Importing "real-world" files into simulation
  - via CD-ROM and via hostfs mounts
- Examining/Hacking Simics checkpoint file

Flexus Simulator Toolset

Michael Ferdman

Computer Architecture Lab at Carnegie Mellon

December 4, 2010

Software Simulation

- Fast and easy to implement
  - Minimal cost, simulator runs on your desktop
  - Reuse components, don't implement everything
- Enables standard benchmarks (SPEC, TPC)
  - Can execute real applications
  - Can simulate thousands of disks
  - Can simulate very fast networks

Main Idea

- Use existing system simulator (Simics)
  - Handles BIOS (booting, I/O, interrupt routing, etc…)
- Build a "plugin" architectural model simulator
  - Fast – read state of system from Simics
  - Detailed – interact with and throttle Simics

Developing with Flexus

- Flexus philosophy
- Fundamental abstractions
- Important support libraries
- Components and what they model

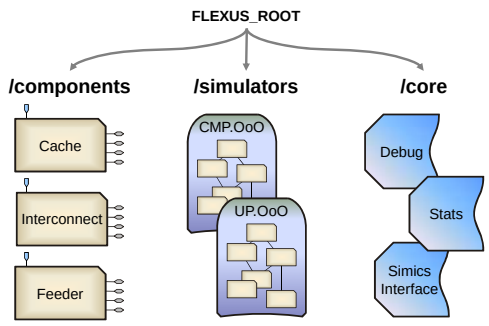
Flexus philosophy

- Component-based design
  - Compose simulators from encapsulated components
- Software-centric framework
  - Flexus abstractions are not tied to hardware
- Cycle-driven execution model
  - Components receive "clock-tick" signal every cycle
- SimFlex methodology
  - Designed-in fast-forwarding, checkpointing, statistics

Developing with Flexus

- Flexus philosophy
- **Fundamental abstractions**
- Important support libraries
- Components and what they model

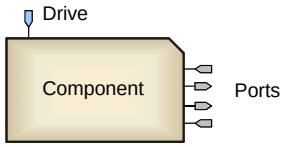
Flexus organization



Fundamental abstractions

- Component
  - Component interface
    - Specifies data and control entry points
  - Component parameters
    - Configuration settings available in Simics or cfg file
- Simulator
  - Wiring
    - Specifies which components and how to connect
    - Specifies default component parameter settings

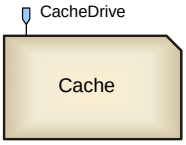
Component interface



- Component interface (terminology inspired by *Asim* [Emer 02])
  - Drive: "clock-tick" control entry point to component
  - Port: specifies data flow between components

Components w/ same ports are interchangeable

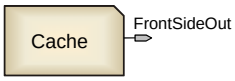
Abstractions: Drive




```
COMPONENT_INTERFACE(
  ...
  DRIVE ( Name )
  ...
);
```

- Control entry-point
- Function called once per cycle

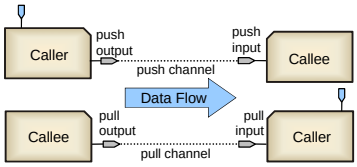
Abstractions: Port




```
COMPONENT_INTERFACE(
  ...
  PORT ( Type, Payload, Name )
  ...
);
```

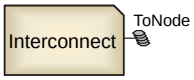
- Data exchange between components
- Ports connected together in simulator wiring

Types of ports and channels



- Type - direction of data and control flow
  - Control flow: Push vs. Pull
  - Data flow: Input vs. Output
- Payload - arbitrary C++ data type
- Type and payload must match to connect ports
- Availability - caller must check if callee is ready

Port and component arrays




```
COMPONENT_INTERFACE(
  ...
  DYNAMIC_PORT_ARRAY(...)
  ...
);
```

- 1-to-*n* and *n*-to-*n* connections
  - E.g., 1 interconnect -> *n* network interfaces
- Array dimensions can be dynamic

Example code using a port

SenderComponent.cpp

```
void someFunction() {
    Message msg;
    if ( FLEXUS_CHANNEL(Out).available() )  {
        FLEXUS_CHANNEL(Out) << msg;
    }
}
```

ReceiverComponent.cpp

```
bool available( interface::In )              { return true; }
void push( interface::In, Message & msg)  { ... }
```

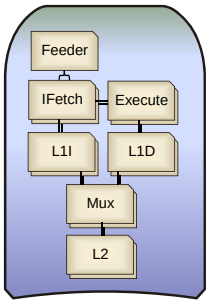

Configuring components

- Configurable settings associated with component
  - Declared in component specification
  - Can be std::string, int, long, long long, float, double, enum
  - Declaration: PARAMETER(BlockSize, int, "Cache block size", "bsize", 64)
  - Use: cfg.BlockSize
- Each component instance associated with configuration
  - Configuration declared, initialized in simulator wiring file
  - Complete name is <configuration name>:<short name>
- Usage from Simics console
  - flexus.print-configuration flexus.write-configuration "file"
  - flexus.set "-L2:bsize" "64"

Simulator wiring

simulators/*name*/Makefile.*name*
- List components for link
- Indicate target support

simulators/*name*/wiring.cpp
1. Include interfaces
2. Declare configurations
3. Instantiate components
4. Wire ports together
5. List order of drives



Developing with Flexus

- Flexus philosophy
- Fundamental abstractions
- **Important support libraries**
- Components and what they model

Critical support libraries in /core

- Statistics support library
  - Record results for use with `stat-manager`
- Debug library
  - Control and view Flexus debug messages

Statistics support library

- Implements all the statistics you need
  - Histograms
  - Unique counters
  - Instance counters
  - etc…
- Example:

```
Stat::StatCounter  myCounter( statName() + "-count" );
++ myCounter;
```


A typical debug statement

```
DBG_(Iface,                                  Severity level
  Comp(*this)                                Associate with this component
  AddCategory( Cache )                       Put this in the "Cache" category
  ( << "Received on FrontSideIn[0](Request): "
    << *(aMessage[MemoryMessageTag])
  )                                          Text of the debug message
  Addr(aMessage[MemoryMessageTag]->address())
);                                           Add an address field for filtering
```


Debug severity levels

1. Tmp — temporary messages (cause warning)
2. Crit — critical errors
3. Dev — infrequent messages, e.g., progress
4. Trace — component defined – typically tracing
5. Iface — all inputs and outputs of a component
6. Verb — verbose output from OoO core
7. VVerb — very verbose output of internals

Controlling debug output

- Compile time
  - `make` *target-severity*
  - *(e.g.* `make UP.Trace-iface`*)*
- Run time
  - `flexus.debug-set-severity` *severity*
- Hint – when you need a lot of detail…
  - Set severity low
  - Run until shortly before point of interest (or failure)
  - Set severity high
  - Continue running

Developing with Flexus

- Flexus philosophy
- Fundamental abstractions
- Important support libraries
- **Components and what they model**

Simulators in Flexus 4.0

- UP.Trace — fast memory system
- CMP.L2Shared.Trace — fast CMP memory system
- UP.OoO — 1 CPU 2-level hierarchy
- CMP.L2SharedNUCA.OoO — private L1 / shared L2
- CMP.L2Private.OoO — private L1 / private L2

Memory hierarchy

- "top", "front" = closer to CPU
- Optimized for high MLP
  - Non-blocking, pipelined accesses
  - Hit-under-miss within set
- Coherence protocol support
  - Valid, modifiable, dirty states
  - Explicit "dirty" token tracks newest value
  - Non-inclusive
  - Supports "Downgrade" and "Invalidate" messages
  - Request and snoop virtual channels for progress guarantees

Out-of-order execution

- Timing-first simulation approach [Mauer 2002]
  - OoO components interpret SPARC ISA
  - Flexus validates its results with Simics
- Idealized OoO to maximize memory pressure
  - Decoupled front end
  - Precise squash & re-execution
  - Configurable ROB, LSQ capacity; dispatch, retire rates
- Memory order speculation (similar to [Wenisch 07])

Hands-on

- Setting up `.run_job.rc.tcl` file
- Launch Simics using the `run_job` script
- Build Flexus simulators
  - Examine Flexus directory structure and source files
- Launch trace-based simulation
- Launch cycle-accurate (OoO) simulation
  - Examine debug output and statistics

How fast is cycle-accurate timing simulation?

SimFlex: Fast, Accurate, and Flexible Simulation of Computer Systems

Pejman Lotfi-Kamran

Anastassia Ailamaki, Babak Falsafi, James Hoe, Kun Gao, Brian Gold, Nikos Hardavellas, Jangwoo Kim, Ippokratis Pandis, Minglong Shao, Jared Smolens, Stephen Somogyi, Thomas Wenisch, Roland Wunderlich

Computer Architecture Lab at Carnegie Mellon

December 4, 2010

Simulation speed challenges

- Longer benchmarks
  - SPEC 2006: *Trillions* of instructions per benchmark
- Slower simulators
  - Full-system simulation: 1000× slower than SimpleScalar

- Multiprocessor systems
  - CMP: 2× cores every processor generation

1,000,000× slowdown vs. HW → years per experiment

The measurement challenge: Slow full-system simulation

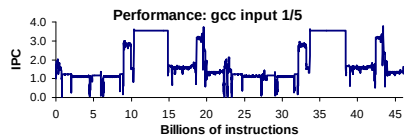
- Simulation slowdown per cpu

– Real HW:	~ 500 MIPS	1 s
– Simics:	~ 15 MIPS	33 s
– Flexus, no timing:	~ 1.5 MIPS	5.5 m
– Flexus, in-order:	~ 10 kIPS	13.8 h
– Flexus, OoO:	~ 3 kIPS	46 h

150 years for 1-CPU audited TPC-C run in OoO simulation

Current simulation practices

- Subset or scaled version of benchmark suite
- Single unit of ~1 billion instructions
- Selected measurements via profiling



Results not representative of workload performance

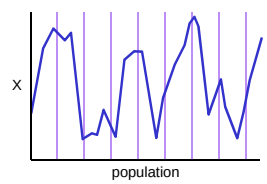
Our Solution: Statistical sampling

- Measure uniform or random locations

- Impact
  - Sampling: ~10,000× reduction in turnaround time
  - Independent measurements: 100- to 1000-way parallelism
  - Confidence intervals: quantified result reliability

Sampling theory

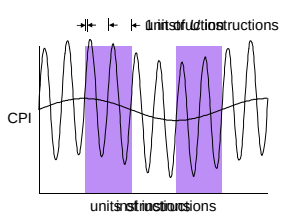
Estimate the mean of a population property *X* — to a desired confidence — by measuring *X* over a sample whose size *n* is minimized.



- Arbitrary distribution
- Confidence
  - e.g., 99.7% probability of ±3% error
- $n = f($ C.V., confidence $)$

Sampling for simulation

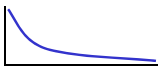
Defining the sampling population



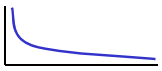
- CPI difficult to measure over 1 instruction
- Instead, define as units of *U* instructions
- As *U* changes, so does:
  - Observed C.V. of CPI
  - Required sample size *n*

Minimizing total instructions

Coefficient of variation V_{CPI}



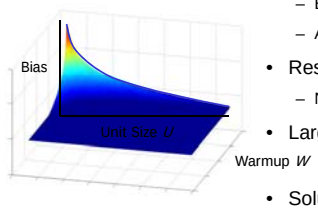
Required sample size n



Total instructions $n \cdot U$

A large sample of small units minimizes total instr.

Small units in practice: Bias



- Inexact simulator state
  - Empty pipeline
  - Approximate caches, etc.
- Results in bias
  - Non-random error
- Larger effect as *U* shrinks
- Solution: Warmup before each unit to correct state

Handling Bias

- For fast results, must measure short pieces
  - If starting from cold state, introduces bias
- Perform "wamup" prior to measurement
  - Functional warming during fast-forwarding
  - Detailed warmup before each simulation window

What is SMARTS runtime problem?

- 99% of runtime spent functional warming
  - Average SPEC CPU2000 ref. input: 170 billion instructions
  - Average SMARTS detailed simulation: 25 million instructions
- Longer benchmarks
  - Similar sample size because similar V_{CPI}
  - More functional warming

Replace functional warming with checkpoints

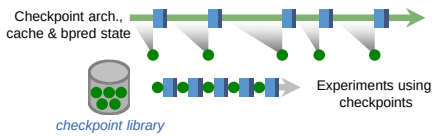
What needs to be checkpointed?

- Functional warming for multiple configurations
  - Same architectural state
  - Different microarchitectural state
- If checkpoints replace functional warming
  - SMARTS accuracy & confidence in results
  - Huge speedup & parallelism
  - Online results & matched-pair comparison

Checkpoint requirements

1. Reusable warm microarchitecture state
   - Cache hierarchy
   - Branch predictor
2. Independent - no sequential dependency
   - Parallelism
   - Sampling optimizations

Warm microarchitecture state



- Store warm cache & branch predictor state
  - Same sample design, accuracy, confidence
  - No warming length prediction needed

Works for any microarchitecture if reusable cache & branch predictor state

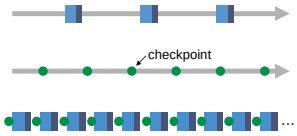
SMARTS & TurboSMARTS

- SMARTS: Sampling Microarchitecture Simulation
  - SMARTSim extends SimpleScalar with sampling
  - SPEC CPU2000 avg. benchmark in 7 hrs vs. 5.5 days

- TurboSMARTS: Live-point support
  - Checkpointed warming enables parallel simulation & online results
  - SPEC CPU2000 avg. benchmark in 91 seconds

The simplified SimFlex procedure

1. Prepare workload for simulation
   – port workload into Simics
2. Measure baseline variance
   – determine required library size
3. Collect checkpoints
   – via functional warming
4. Detailed simulation
   – estimate performance results



1. Preparing a workload for simulation

- Bring application up in Simics
  - Install OS, application
  - Construct workload (e.g., load DB)
  - Prepare disk images
  - Tune workload parameters

2. Determining sampling parameters

- Can't switch modes during simulation
  - Simics runs in different modes for each timing model
- Instead, construct preliminary flex-point library
  - 30-50 flex-points to estimate C.V., detailed warming
  - Good initial guess: C.V. for user IPC ≈ 0.5

Typical sampling parameters

	SMARTSim (8-way OoO)	Flexus (16-CPU CMP.OoO)
Warming	2000 inst.	100k cycles
Measurement	1000 inst.	50k cycles
Target confidence	99.7% ± 3%	95% ± 5%
Sample size	8000	200-400
Sim. time per checkpoint	10 ms	~ 5 min
Experiment turnaround time	91 CPU-sec	~ 5 CPU-hours

3. Collect checkpoints

- Single run too slow
- Need multi-tier approach optimized to:
  - Leverage speed of Simics "fast" mode
  - Parallelize flex-point creation across CPUs
  - Minimize storage
  - Manage dependence between flex-point files

Constructing flex-points

- Use *UP.Trace or CMP.Trace*
  - High speed (1.5 MIPS)
    - 100 insn / CPU Simics simulation quantum
    - Minimal code paths to collect cache, bpred state
  - Fully deterministic
    - No timing feedback into Simics
    - i.e., execution in Simics as if Flexus not present

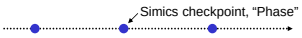
Simics checkpoints

- Simics requires complete arch. state
  - Full checkpoints
    - Proportional to system RAM, ~2GB
    - Independent
  - Delta checkpoints
    - Size proportional to memory updates; 50-300MB
    - Dependant – requires all files in chain
    - Unix file descriptors limit delta chain length (~25)

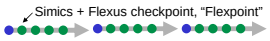
Storage constraints limit sample sizes to 100's

Flex-point creation timeline

1. Spread Simics checkpoints
   - via Simics -fast
   - rapidly cover 30s



2. Collect flex-points in parallel
   - via *UP.Trace*
   - From each Simics checkpoint



4. Detailed simulation

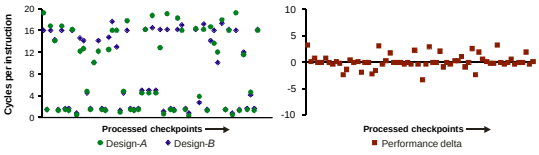
- Process all flexpoints, aggregate offline
- Manipulate results & stats with `stat-manager`
  - Each run creates binary `stats_db.out` database
  - Offline tools to select subsets; aggregate
  - Generate text reports from simple templates
  - Compute confidence intervals for mean estimates

Matched-pair comparison [Ekman 05]

- Often interested in relative performance
- Change in performance across designs varies less than absolute change
- Matched pair comparison
  - Allows smaller sample size
  - Reports confidence in performance change

Matched-pair example

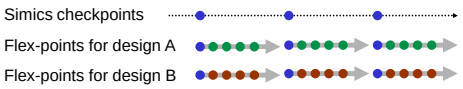
Performance results for two microarchitecture designs
checkpoints processed in random order



Lower variability in performance deltas reduces sample size by 3.5 to 150×

Matched-pair with Flexus

- Simple µArch changes (e.g., changing latencies)
  - use same flex-points
- Complex changes (e.g., adding components)
  - use aligned flex-points



  - *.Trace* simulators are fully deterministic
    - Produces identical insn. stream across simulations
    - Flex-points can share Simics state

Hands-on

- Generate Flexpoints
- Launch timing simulation for all Flexpoints
- Aggregate stats with `stat-collapse`
- Examine aggregated statistics
  - Compute confidence
  - Plot timing breakdowns

SimFlex: Conducting Research with Flexus

Michael Ferdman

Computer Architecture Lab at Carnegie Mellon

Conducting Research with Flexus

- Workload statistics collection
- Design implementation and tuning
- Flexpoint generation
- Timing evaluation

Simple Example: Victim Cache Workload statistics collection

- Instrument cache to count Conflict Misses
  - components/FastCache/FastCacheImpl.cpp
- Collect baseline statistics

```
run_job –cfg victim –run trace UP.Trace oracle
```

- Review results

```
stat-manager format victim.rpt
```

victim.rpt contents:

```
L2 D-cache misses: <EXPR:sum{.*L2-Misses:User:D:Read}>
L2 D-cache conflict misses: <EXPR:sum{.*L2-Misses:User:D:Read:Conflicts}>
```


Simple Example: Victim Cache Design implementation and tuning

- Model victim cache in trace simulation
  - components/FastCache/FastCacheImpl.cpp
- Tune the design (sizes, replacement policy)

```
run_job –cfg victim-8-lru –run trace UP.Trace oracle
run_job –cfg victim-16-rnd –run trace UP.Trace oracle
```

- Confirm intuition and select design
  - Can reuse victim.rpt for `stat-manager`

Simple Example: Victim Cache FlexState generation

- Implement checkpoint save/restore in UP.Trace
  - components/FastCache/FastCacheImpl.cpp
- Implement checkpoint restore in timing
  - components/Cache/CacheControllerImpl.cpp
- Generate FlexState for design with victim cache

```
run_job \
  -postprocess "postprocess_ckptgen.sh flexpoint 10 vict-8-lru" \
  -cfg vic-8-lru -run flexpoint -local UP.Trace oracle
```


Simple Example: Victim Cache Timing Evaluation

- Run timing jobs

```
run_job -cfg victim -run timing UP.OoO oracle
```

- Use `stat-collapse` to select measurements
- Use `stat-sample` to compute speedup
  - generate *sets* of UIPC numbers (baseline and victim)
  - matched-pair comparison on UIPCs

