**Simulation speed challenges**

- **Longer benchmarks**
  - SPEC 2006: *Trillions* of instructions per benchmark

- **Slower simulators**
  - Full-system simulation: $1000\times$ slower than SimpleScalar

- **Multiprocessor systems**
  - CMP: $2\times$ cores every processor generation

\[ 1,000,000\times \text{slowdown vs. HW} \rightarrow \text{years per experiment} \]
Our solution: Statistical sampling

- Measure uniform or random locations

- Impact
  - Sampling: $\sim 10,000\times$ reduction in turnaround time
  - Independent measurements: 100- to 1000-way parallelism
  - Confidence intervals: quantified result reliability

- Challenges
  - Rapidly create warm microarch state prior to measurements
  - Allow independent simulation of each measurement
  - Sample non-deterministic, highly variable MP applications

SimFlex tutorial

- Simulation sampling
  - Background & theory
  - Best practices

- Flexus full-system simulation framework
  - Simulator usage & multiprocessor sampling
  - Developing with Flexus

**Best practices for simulation-based multiprocessor research**
Tutorial outline

- Introduction (15 min)
- Simulation sampling (45 min)
- Multiprocessor workloads on Flexus (1 hour)
- Developing with Flexus (1 hour)

Tutorial materials

- Hand-outs
  - Bound set of this tutorial’s slides
  - ‘Getting Started with Flexus’ quick reference

- www.ece.cmu.edu/~simflex
  - TurboSMARTS, Flexus downloads
  - Publications, tech reports
  - Discussion/Q&A mailing list
SimFlex terminology

- SimFlex research group
  - SMARTS sample design
    - SMARTSim, TurboSMARTS simulators
  - Flexus full-system simulation framework
    - TraceFlex, UniFlex, DSMFlex, etc. simulators

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
Flexus framework

- Full-system simulation of unmodified commercial apps
  - In-order timing on any Simics target
  - OoO timing for SPARC v9
- Component-based design
  - Multiple timing modes trade accuracy for speed
  - Easy composition of complex system models
- Uniprocessor, CMP, DSM system models
  - CMP, DSM hardware models based on Piranha
  - Aggressive OoO core tuned to produce high memory parallelism
- Designed-in support for simulation sampling
  - Checkpointing of arch and μarch state
  - Statistic aggregation and confidence calculations

Flexus timing modes

- No timing (e.g., TraceFlex) ~1.5 MIPS
  - High speed functional simulator
  - For trace studies and checkpoint creation
- In-order timing (e.g., UniFlex) ~10 kIPS
  - Simics blocked while Flexus times a memory op
  - Easily modified to support any Simics target
- Out-of-order timing (e.g., UniFlex.OoO) ~3 kIPS
  - Timing first simulation approach [Mauer 02]
  - SPARC v9 ISA
Related simulators

- GEMS [U. Wisconsin]
  - Strength: coherence protocols & optimization

- M5 [U. Michigan]
  - Strength: network & I/O integration

- Liberty [Princeton]
  - Strength: structural modeling

*Flexus’ focus: component-based design*

Flexus usage at Carnegie Mellon

- STEMS – Spatio-Temporal Memory Streaming
- TRUSS – Total Reliability Using Scalable Servers
- ProtoFlex – Hardware prototype component verification
- Staged DB/CMP – Pipelining DB operators across CMP
- Graduate computer architecture classes
Tutorial outline

• Introduction
• Simulation sampling
  • Multiprocessor workloads on Flexus
  • Developing with Flexus

Simulation Sampling – Background & Theory

SimFlex Tutorial – Section 2 of 4

Roland Wunderlich
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

Uniprocessor simulation sampling

- Measure many units of few instructions
  - Representative results with minimal simulation
  - No profiling for uniform sampling: immune to many types of error
- Functional warming – update state between units
  - Enables accurate measurement of small units
- Live-points – checkpointed warming
  - Enables more speed, parallelism, and online results
- SMARTSim & TurboSMARTS results
  0.6% CPI error, 20× & 5000× speedup on SPEC CPU2000
**Section 2 outline**

- **Sampling in theory**
  - Sampling theory recommendations (assuming accurate measurements)

- **Sampling in practice**
  - Accurately measuring small units
    - Online warming: functional warming
    - Checkpoints: live-points

---

**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
- $n = f(\text{C.V., confidence})$
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.
Variation of CPI in SPEC CPU2000

- Optimal unit size $U \leq 1000$ instructions
- $V_{CPI}$ clustered around 1
  \[ \therefore \text{sample size } n \approx 10,000 \text{ likely to give } 99.7\% \pm 3\% \text{ confidence} \]

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
**Warmup requirements of SPEC 2000**

- Warmup requirements vary **widely and unpredictably**

![Graph showing warmup required to achieve ≤ ±1.5% Bias](image)

**Addressing unpredictable warmup**

<table>
<thead>
<tr>
<th>Type of history</th>
<th>Warming strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short, predictable</td>
<td>Short detailed warmup</td>
</tr>
<tr>
<td>• ROB, pipeline registers</td>
<td>• Bound via worst-case analysis</td>
</tr>
<tr>
<td>Long, unpredictable</td>
<td>Functional warming</td>
</tr>
<tr>
<td>• Caches, branch predictors</td>
<td>• Update state between units</td>
</tr>
<tr>
<td>Long, requires exact timing</td>
<td>Analytic modeling</td>
</tr>
<tr>
<td>• HW interrupt timing</td>
<td>• No typical superscalar structures in this category</td>
</tr>
</tbody>
</table>

**SMARTS**

- Functional warming
- Detailed warmup
- Measurement
Effectiveness of functional warming

Ten worst bias results
Average bias over 5 experiments with 2000 inst. detailed warmup

Functional warming reduces bias below 2% in all cases

Optimal sampling parameters with SMARTS warming strategy

Percent of benchmark detail-simulated \( n \cdot (U + W) \)
\( W = 2000; \ n \) chosen for 99.7% ± 3% confidence

Detailed simulation is nearly negligible (< 50 million instr.)
**SMARTS accuracy results**

CPI Error Results
with recommended initial $n = 10,000$ $U = 1000$ $W = 2000$

Desired confidence can be achieved by rerunning experiments with higher $n_{tuned}$ calculated from the observed $V_{CPI}$

Worst-case error 2.4%, avg. 0.6%, 20× speedup

---

**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. Small & fast loading
   - Average SPEC CPU2000 memory footprint: 105 MB
   - Must process in < 2 sec/checkpoint to improve SMARTS

3. 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*

Reusable cache state

- Goal – reconstruct multi-configuration warm caches
  - Replay minimal memory trace into cache
  - Concurrently developed: Memory Timestamp Record [Barr 2005]

- Checkpoint creation
  - Simulate maximum interesting cache size, associativity
  - Track cache line access times
  - Store time ordered list of cache-resident memory addresses

- Checkpoint usage: replay memory trace
Remaining problems

- Reusable branch predictor state
  - Current solution: store multiple warm branch predictor configs
  - Branch trace compression [Barr 2006]
    • Efficiently stores branch outcome trace to train predictor

- Main memory data size
  - Average SPEC CPU2000 memory footprint: 105 MB
  - Results in storage problem & slow loading checkpoints
  - Store non-speculative memory subset, complete cache tags

Independent checkpoints

- 100- to 1000-way parallelism
  - Distribute checkpoints across compute cluster
  - No need to multithread detailed simulator

- Online results
  - Report results while simulation in progress
  - Results converge as checkpoints are processed

- Matched-pair comparison
  - Comparative experiments need smaller sample
  - Provides confidence in performance delta
## Uniprocessor sampling results

<table>
<thead>
<tr>
<th></th>
<th>SimpleScalar</th>
<th>SMARTSim</th>
<th>TurboSMARTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average (worst) CPI bias</td>
<td>None</td>
<td>0.6% (1.6%)</td>
<td>0.6% (1.6%)</td>
</tr>
<tr>
<td>Average benchmark runtime</td>
<td>5.5 days</td>
<td>7.0 hours</td>
<td>91 seconds</td>
</tr>
<tr>
<td>Parallel sim. &amp; online results</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>SPEC CPU2000 chpt. Library</td>
<td></td>
<td></td>
<td>12 GB</td>
</tr>
<tr>
<td>Fixed microarch. parameters</td>
<td></td>
<td></td>
<td>Max cache, TLB, branch predictors</td>
</tr>
</tbody>
</table>

*Simulation sampling + live-points: Accurate, faster, enables sampling optimizations*

## Simulation sampling conclusions

- **Live-points** – small, fast loading, reusable checkpoints

- Simulation sampling + checkpoints
  - Results with confidence intervals
  - Huge speedup & parallelism
  - Online results & matched-pair comparison

- **TurboSMARTS** available at SimFlex website
  - Approaches uni-processor simulation speed limit
MP sampling needs checkpoints

- Simulation sampling + checkpoints is essential
  - Full-system simulation
    - 1000× slower than SimpleScalar
    - More complex warming needs
  - Commercial server benchmarks
    - Multiprocessor + OS & I/O intensive
    - Performance often measured with transaction throughput

**Flexus**: designed-in support for sampling + “flex-points”

Tutorial outline

- Introduction
- Simulation sampling
- **Multiprocessor workloads on Flexus**
  - Developing with Flexus
What makes server apps harder?

- Server application characteristics
  - multi-threaded, multi-processor
  - large memory footprint
  - I/O intensive
  - OS performance matters
  - client-server
  - complicated workload setup & tuning
  - non-deterministic behavior

SPEC CPU has none of these characteristics
Functional simulation challenges

- Functional simulator must support
  - multiple CPUs
  - networks of systems (client-server)
  - privileged-mode ISA
  - peripheral devices
  - more RAM than host system
  - saving/restoring architecturally visible state

Simics provides these capabilities

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
Section 3 outline

- Extending sampling to MP applications
- The SimFlex experiment procedure
- Sampling optimizations

Our approach in this tutorial

- Develop sampling approach for MP
  - Use SMARTS, Live-points as a starting point
  - Re-examine each step in the MP context

- Primary focus: throughput applications
  - E.g., transaction processing, web serving
  - Open problems remain for general MP apps.
**MP sampling challenges**

- Non-deterministic & interleaved instruction streams
  - Uniprocessor population definition inappropriate
  - Define population in terms of possible interleavings
- Long & highly variable transaction latency
  - Unacceptable sample & measurement sizes
  - Measure fine-grain progress metrics
- Complex, inter-related queues (e.g., interconnect)
  - Complicated detailed warmup analysis
  - Empirical queue warmup detection

---

**SMARTS summary**

- SMARTS Sample Design
  - Population: \( N \) units of \( U \) instructions
  - Performance metric: CPI
  - Detailed warming: \( W \) worst-case analysis: \(~2000\ \) inst.
  - Optimal unit size: \( U \) from \( V_{CPI} \): \(~1000\ \) inst
  - Typical sample size: \( n \) \(~8000\ \) inst

Revisit each aspect for MP applications
Population

- **SPEC**: \( N \) units of \( U \) instructions each
- **Servers**: Instruction stream is not fixed
- Characteristics of server instruction stream
  - Unbounded
  - Non-deterministic
  - Multiple parallel streams

SPEC definition for population inappropriate

---

**MP population definition**

- Real HW pop.: repeat/extend runs for stable results
  - Non-determinism & interleaving problematic
  - Achieve stable results by *observing many interleavings*
- Sampling pop.: set of reachable states
  - Throughput apps – *random transaction arrivals*
  - Determine minimum run length on real HW (e.g., 30s)
  - Draw sample of reachable states from such a run
Constructing a sample in simulation

- Pop. defined in terms of states reachable in OoO
- However, construct sample via fn. warming
  - May give unrepresentative interleaving of start PCs

Random transaction arrivals – any PC positions possible
Open problem for non-throughput applications

Performance Metric

- **SPEC**: Cycles per instruction (CPI)
- **Servers**: CPI is not proportional to performance
  - Not all instructions make forward progress
  - Frequent spins on I/O and locks

Desired metric characteristics
  - Proportional to forward progress
  - Responds quickly to performance change
  - Low variance at small unit sizes
**MP performance metric**

- Typical metric: Transactions completed / min. (TPM)
- Coarse progress metrics bad for sampling
  - Transaction completions infrequent
  - High variance of inter-arrival and service times

Can’t assess TPM reliably with small (<10M inst.) window

**Measure fine-grain progress**

- User-mode inst. per trans. constant per app. cfg.
- Hence, user-mode IPC \(\propto\) TPM
  - Validated for TPC-C, SpecWEB [Hankins 2003, Wenisch 2006]
  - Most apps yield rather than spin in user mode

Impact: Same confidence with 1000x shorter measurements
Detailed warming ($W$)

- **SPEC**: Worst-case analysis; ~2000 instructions
- **Servers**: Worst-case analysis is difficult
- Large queues complicate analysis
  - Miss handlers in memory system
  - Router buffers in interconnect
- Unlike caches, queue state cannot persist indefinitely
  - Queue warming brief, bounded
  - Results in steep slope in cold-start bias

Detect steep bias slope empirically

Empirical queue warming analysis

- Bias - relationship btw. perf. and time into msmt.
  - Determine minimum time where bias disappears

![Performance vs. Distance into Measurement](sample size = 50)
Unit size ($U$)

- **SPEC**: Examine tradeoff of $U$ and total sim.
- **Servers**: Same approach
  - Higher warmup requirement
  - Lower CV (summing over CPUs smoothes variation)

![Graph showing total simulation for +/- 5%](image)

Section 3 outline

- Extending sampling to MP applications
- The SimFlex experiment procedure
- Sampling optimizations
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

Procedure with sampling optimizations later
Preparing Simics disk images

• Best approach: collect image from real system
  – Install & configure workload on real system
  – Collect images of disk partitions with `dd` & `craff`
  – See `Simics User Guide §8`

• Alternative: install in simulation
  – See `Simics Serengeti Target Guide §6`
  – Not recommended for commercial server software

Up to 30GB per workload for disk images

2. Determining sampling parameters

• SMARTSim: Use a preliminary SMARTS run

• Flexus: 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

<table>
<thead>
<tr>
<th></th>
<th>SMARTSim</th>
<th>Flexus</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(8-way OoO)</td>
<td>(16-CPU DSMFlex.OoO)</td>
</tr>
<tr>
<td>Warming</td>
<td>2000 inst.</td>
<td>100k cycles</td>
</tr>
<tr>
<td>Measurement</td>
<td>1000 inst.</td>
<td>50k cycles</td>
</tr>
<tr>
<td>Target confidence</td>
<td>99.7% ± 3%</td>
<td>95% ± 5%</td>
</tr>
<tr>
<td>Sample size</td>
<td>8000</td>
<td>200-400</td>
</tr>
<tr>
<td>Sim. time per checkpoint</td>
<td>10 ms</td>
<td>&lt; 20 min</td>
</tr>
<tr>
<td>Experiment turnaround time</td>
<td>91 CPU-sec</td>
<td>~ 30 CPU-hours</td>
</tr>
</tbody>
</table>

3. Collect checkpoints

- SMARTSim: One run with `sim-mkckpt`
- Flexus: Single run too slow to cover 30s
- 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 TraceFlex or TraceCMPFlex
  - 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
1. Spread Simics checkpoints
   - via Simics -fast
   - rapidly cover 30s

2. Collect flex-points in parallel
   - via TraceFlex
   - From each Simics checkpoint

4. Detailed simulation
   - SMARTSim: Process live-points & print results
   - Flexus: Process all flex-points, 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
Section 3 outline

- Extending sampling to MP applications
- The SimFlex experiment procedure
- Sampling optimizations

Sampling optimizations

- Online results
  - Report results while simulation in progress
    - Process precise sample size for target confidence
    - Rapid feedback to detect “broken” experiments
- Matched-pair comparison
  - Reduced sample size for comparative experiments
    - Faster turnaround
    - Lower storage requirements
Online results

- Checkpoints can be processed in arbitrary order
  - Random subset of checkpoints forms valid sample
  - Simulating in random order allows online results

As checkpoints are processed, results converge toward their final values & confidence improves

Randomizing a checkpoint library

- **SMARTSim**: Shuffle live-points in advance
  - Individual live-points ~40KB compressed
  - Sequential I/O improves sim. speed

- **Flexus**: Shuffle flex-points on use
  - Individual flex-points 10-200MB compressed
  - I/O perf. unaffected by shuffling in advance
**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*

Processed checkpoints

![Graph showing processed checkpoints for two microarchitecture designs processed in random order.](image)

*Lower variability in performance deltas reduces sample size by 3.5 to 150×*
Matched-pair with Flexus

- Simple $\mu$Arch changes (e.g., changing latencies)
  - use same flex-points
- Complex changes (e.g., adding components)
  - use aligned flex-points
  - TraceFlex is fully deterministic
    - Produces identical insn. stream across simulations
    - Flex-points can share Simics state

The complete SimFlex procedure

1. Prepare workload for simulation
   - port workload into Simics
2. Measure baseline variance
   - determines needed library size
3. Collect checkpoints
   - functional warming used
4. Shuffle live-point library
   - randomize for online results
5. Baseline experiment
   - estimates absolute perf.
6. Matched-pair experiments
   - estimates comparative perf.
SimFlex results

<table>
<thead>
<tr>
<th></th>
<th>SPEC 2000 (per input)</th>
<th>Commercial Apps. (OLTP, DSS, Web)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Measurement time on hardware</td>
<td>2 min.</td>
<td>30 sec.</td>
</tr>
<tr>
<td>Simulation time without sampling</td>
<td>4.5 CPU-days</td>
<td>10-20 CPU-years</td>
</tr>
<tr>
<td>Time with SimFlex approach</td>
<td>91 CPU-sec.</td>
<td>70 CPU-hours</td>
</tr>
<tr>
<td>Storage for checkpoints</td>
<td>273 MB</td>
<td>30 GB</td>
</tr>
</tbody>
</table>

SimFlex approach makes simulation studies tractable

Tutorial outline

• Introduction
• Simulation sampling
• Multiprocessor workloads on Flexus
• Developing with Flexus
Developing with Flexus

SimFlex Tutorial – Section 4 of 4

Thomas Wenisch

- Flexus philosophy
- Fundamental abstractions
- Important support libraries
- CMU 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

Component-based design

Simulators assembled from composable parts

- Motivation
  - Flexibility
  - Encapsulation
  - Model refinement
  - Parallel development
- Implementation
  - Explicit interface specs
  - Separate source code
Software-centric framework

Motivation
- Behavioral modeling not structural
- SW-only components e.g., trace generators

Implementation
- High-level components “Cache”; not “Sub-array”
- Zero-latency connections

Cycle-driven execution

Components receive “clock-tick” every cycle

Motivation
- Component isolation
- Cooperating FSMs

Implementation
- “Drive” interfaces
- Specified clocking order

Some components event-driven internally
- E.g., interconnect component
SimFlex methodology

Flexus designed to support simulation sampling

- Motivation
  - Measure MP apps
  - Leverage Simics speed

- Implementation
  - Flex-points
  - stat-manager

Developing with Flexus

- Flexus philosophy

- Fundamental abstractions

  - Important support libraries

  - CMU components and what they model
**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

- 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**

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

**ReceiverComponent.cpp**

```cpp
bool available( interface::In ) { return true; }
void push( interface::In, Message & msg ) { … }
```
Component-based design & payloads

• Problem: new comps add fields to messages

Bad solution: modify struct for message
  – Breaks compatibility w/ existing components
  – Leads to conflicts in header files

Transports

• Scalable data structure
  – Made up of one or more slices
  – Transport object holds pointers to slices
  – Reference counting to manage memory
  – Implemented via templates (typesafe; low overhead)
  – Syntax similar to array access

Adding a new slice has no effect on existing code
Reference counted pointers

- Used for object exchange between comps.
  - Reduces need to think about object lifetime
- E.g., ArchitecturalInstruction object
  - Created in InorderSimicsFeeder component
  - Preserved while any comp. references instruction
  - Destroyed once last component “forgets” instruction
- Implementation: Boost intrusive pointer
  - Documentation: www.boost.org/libs/smart_ptr

Simulator wiring

- List components for link
- Indicate target support

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

- Wiring implemented via C++ overload resolution
  - Push-out, pull-in port accesses call stub functions
  - Stub functions emitted when compiling wiring.cpp
- Advantages
  - Type safety
  - Linker resolves component interconnection
  - Constant time; independent of # wires, # components

Developing with Flexus

- Flexus philosophy
- Fundamental abstractions
- Important support libraries
  - CMU 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

- Key features
  - Support tuning of warming, measurement windows
  - Generate text reports from stats database
  - Calculate formulas & confidence intervals
  - Histograms, unique counters, instance counters

- Example:
  Stat::StatCounter myCounter( statName() + "-count" );
  ++ myCounter;
Flexus statistics collection model

- Multiple concurrent stat collection windows
- Can’t ask for “current value” of a stat
- Measurement windows controlled via:
  - Cycle, transaction, instruction counts
  - Programmatically in Flexus code
  - Events in the simulated system (MagicBreak component)

stat-manager

- Generates reports using report templates
  - Text files w/ placeholders
- Example placeholders:
  - {name}
  - {name@Region 002}
  - {histogram-name;val:2}
  - <EXPR:2*{name}>
  - <EXPR:sum{regular expression}>
Aggregating sample results

- **stat-collapse**
  - Aggregate measurement windows from one stats DB

- **stat-sample**
  - Reads DBs produced by *stat-collapse*
  - Produces sum, avg, stdev, count in output DB
  - 95% confidence interval with `<EXPR:ci95(name)>`

Debug support library

- Filtering at compile time and runtime
  - Via make command line by severity
  - Via simics console by severity, component, category
  - Via debug.cfg files by any field

- Formatting, output to multiple destinations
  - Via debug.cfg files

- Implementation: preprocessor macros
A typical debug statement

DBG_(Iface,
    Comp(*this)
    AddCategory( Cache )
    ( "Received on FrontSideIn[0](Request): "
      "(aMessage[MemoryMessageTag])"
    )
    Addr(aMessage[MemoryMessageTag]->address())
);
Simulators in Flexus 2.1.0

- UniFlex [.OoO] 1 CPU 2-level hierarchy
- CMPFlex [.OoO] private L1 / shared L2
- DSMFlex [.OoO] micro-coded coherence
- TraceFlex Uni or DSM live-points
- TraceCMPFlex CMP live-points

OoO variants support v9, all others support v9/x86

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
**CMP memory hierarchy**

- Shared L2 based on Piranha
  - L2 acts as L1 victim cache
  - L2 maintains directory, duplicate tags
  - Both L1I and L1D cache coherent
  - Private L1’s behave as in uniprocessor hierarchy

**Distributed shared memory system**

- Micro-coded coherence protocol engines
  [contributed by Andreas Nowatzyk]
  - Protocol, engine design, microcode based on Piranha
  - See Piranha publications for overview
- Point-to-point switched interconnect
  - User specifiable interconnect topology, link BW
  - Currently supports any source routed algorithms
  - Deadlock freedom via escape virtual channels
In-order execution

- Assumes one cycle for non-memory instructions
- Supports TSO memory model under SPARC
  - Stores retire to store buffer
  - Store buffer drained in order with store prefetching

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
  - Data flow execution (no functional unit constraints)
  - Precise squash & re-execution
  - Configurable ROB, LSQ capacity; dispatch, retire rates
- Memory order speculation (similar to [Gniady 99])
Reference slides

- Flexus component inventory
- Configuring components
- Debug system details
- Controlling Flexus in code
- Flex-point implementation
- Live-point implementation
- Workload scaling: DBmbench
- Citations
- SimFlex team

Flexus component inventory (1)

- BPWarm – branch predictor for checkpoint creation
- Cache – cache implementation for timing simulation
- CMPCache – shared L2 cache component for timing simulation
- Common – definitions for slices, transports, other shared code
- DecoupledFeeder – interacts with Simics for non-timing simulation
- Directory – DSM directory state storage
- Execute – in-order timing instruction execution unit
- FastBus – coherence/snooping bus for non-timing simulation
- FastCache – non-CMP cache for non-timing simulation
Flexus component inventory (2)

- FastCMPCache – CMP cache for non-timing simulation
- FastMemoryLoopback – main memory for non-timing simulation
- FetchAddressGenerate – branch predictor for OoO simulation
- IFetch – instruction fetch unit for in-order simulation
- InorderSimicsFeeder – interacts with Simics for in-order simulation
- LocalEngine – manages local memory interaction in DSM
- MagicBreak – intercepts magic breakpoints & simulated sys. events
- MemoryLoopback – main memory for timing simulation
- MemoryMap – controls assignment of memory pages to DSM nodes

Flexus component inventory (3)

- MTManager – coordination component for fine-grain multi-threading
- NetShim – interconnect simulator for DSM
- Nic – network interface for connecting to NetShim
- ProtocolEngine – micro-coded DSM coherence protocol engines
- TraceTracker – constructs event traces for cache components
- uArch/v9Decoder – out-of-order core implementation
- uFetch – fetch unit and L1 I-cache for OoO timing
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.Associativity`
- 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"

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`
- Run time
  - `flexus.debug-set-severity severity`
  - `flexus.debug-enable-component component idx`
  - `flexus.debug-enable-category category`
- Output
  - Filter on any field via `debug.cfg`
  - See `debug.cfg` and docs in distribution.
  - BNF in `core/debug/parser.cpp`

Controlling Flexus in code

- `theFlexus` object – overall simulation control
- Controls simulation, interaction with Simics
  - Get cycle number: `theFlexus->cycleCount()`
  - Stop simulation: `theFlexus->terminateSimulation()`
  - see `core/flexus.hpp`, `core/flexus.cpp`
- Add commands to Simics console interface
  - see `Flexus_Obj::defineClass()` in `flexus.cpp`
Components’ flex-point support

• Store μarch state alongside Simics checkpoints

• Store only non-transient state
  – Global “quiesce” flag halts Flexus instruction fetch
  – Each comp. queried to see if it has transient state
  – Save after transient state has drained

• Eases state portability across timing models
  – OoO and trace components share format

Flex-point example:
Cache component

• Permanent state (saved in live-point):
  – Tag array contents
  – Cache line state
  – Replacement order

• Transient state (drained before save):
  – Miss status register contents
  – Input/output request queue contents

  All in-flight memory ops drained during quiesce
Uncompressed live-point contents

- Live-state stores in each live-point
  - Touched memory data from committed instruction stream
  - Complete cache tag arrays, approximates wrong-path schedule

Identifying live-state subset

- At checkpoint creation time
  - Touched subset known only for committed instruction stream
  - Not known for wrong-path (speculative) instructions

- Effects of live-state on simulation
  - Wrong-path instruction latency affects scheduling
    - Pipeline resource contention
  - Wrong-path operand values rarely affect instruction throughput

*We store only state required for correct path execution, approximate wrong-path scheduling*
An Orthogonal Approach: Workload Scaling

• Reduce simulation time by simplifying workload
  – Reduces performance variance
  – Single measurement may capture reachable states

• Macro perf. often dominated by common cases
  – Create simple workloads that stress common case
    – A.k.a. Micro-benchmarks

• Caveat 1: Scaling must not alter behavior
• Caveat 2: Results do not represent full workload

DBmbench

• Database μbmarks for μarch research
  – Identify dominant DB operators in TPC-C and TPC-H
  – Tune simple queries to produce same μArch behavior
  – Validated using HW perf. counters on Pentium III

• Key ideas for successful scaling
  – Scale for a specific goal – e.g., μarch behavior
  – Validate relevant metrics – e.g., miss & CPI breakdown
Citations


• Complete list on SimFlex website

SimFlex developers

• Faculty
  – Anastassia Ailamaki
  – Babak Falsafi
  – James Hoe

• Students
  – Eric Chung
  – Brian Gold
  – Nikos Hardavellas
  – Jangwoo Kim
  – Ippokratis Pandis
  – Minglong Shao
  – Jared Smolens
  – Stephen Somogyi
  – Tom Wenisch
  – Roland Wunderlich

Flexus developers

SMARTS research

• Alumni
  – Shelley Chen

© 2006 T. F. Wenisch, R. E. Wunderlich