Welcome Everybody

Almutaz Adileh
Cansu Kaynak
Pejman Lotfi-Kamran
Stavros Volos

CloudSuite on Flexus

- CloudSuite: Suite for scale-out datacenter services
- Flexus: Fast, accurate & flexible architectural Simulator
- Now, CloudSuite Simics images (up to 64 cores)
- The tutorial is interactive
  - Please ask questions anytime during tutorial
CloudSuite on Flexus tutorial, 9th June 2012, ISCA-39

Agenda

- CloudSuite benchmarks overview
- Full-system simulation with Simics
- Flexus internals and hands-on
- Fast simulation via statistical sampling
- Using CloudSuite images

CloudSuite: A Suite for Emerging Scale-out Applications

Almutaz Adileh
Clouds are Scale-out

- Cloud computing is pervasive
  - User base growing exponentially
  - New services appearing daily

- Serving a global-scale audience requires scaling-out
  - Distribute data and computation to many servers

- How to characterize popular scale-out applications?

Need scale-out benchmarks

Which Benchmarks to Use?

- Benchmarks designed for scale-up

Don’t represent scale-out applications
Key Scale-Out Characteristics

- Serve independent requests/tasks
- Operate on huge dataset split into shards
- Communicate infrequently

CloudSuite 1.0 Overview

Covers popular scale-out services
Data Analytics

- Service fast data generation rates (Big Data)

- Extract useful information from data
  - Predict user preferences, opinions, behavior
  - Benefit from information (e.g., business, security)

- Several examples
  - Book recommendation (Amazon)
  - Spyware detection (Facebook)
  - Photo interestingness (Flickr)

Data Analytics Benchmark

- **Application**: Text classification
  - Sentiment Analysis
  - Spam Identification

- **Software**: Mahout (Apache)
  - Popular MapReduce machine learning library

- **Dataset**: Wikipedia English page articles
Data Analytics Benchmark

• Build a model from a Wikipedia training input
• Master sends Wikipedia documents for classification
• Slaves classify documents locally using model
• Slaves send results to master

CloudSuite 1.0

• Data Analytics
• Data Serving
• Media Streaming
• SW Testing
• Web Search
• Web Serving
Data Serving

- Global-scale online services rely on NoSQL datastores
  - Inherently scalable
  - Suitable for unpredictable schema changes
- Scale out to meet service requirements
  - Accommodate fast data generation rate
  - Provide snappy service for concurrent users

Data Serving Operation

Service User

Frontend

Make reservation

Read Req.

Write Req.

Backend

NoSQL DB

Data Serving Benchmark
Data Serving Benchmark

- Yahoo! benchmark driver
  - Predefined mixes of read/write operations
  - Popularity of access distributions (e.g., zipfian)
  - Interface to popular datastores (e.g., Cassandra, HBase)

Data Serving Benchmark

- Cassandra datastore
  - Popular NoSQL: many use cases (e.g., Expedia, eBay, Netflix)
- Driver generates dataset
  - Defines number & size of fields
  - Populates datastore
CloudSuite 1.0

- Data Analytics
- Data Serving
- Media Streaming
- SW Testing
- Web Search
- Web Serving

Media Streaming

- Media streaming expected to dominate internet traffic

- Increasing popularity of media streaming services
  - Video sharing sites, movie streaming services, etc.
Media Streaming Operation

Service User → Media Server
Establish connection → Play
Teardown

Media Streaming Benchmark

- Implements client-side RTSP communication
- Uses Faban traffic generator
- Allows a flexible mix of requests
  - Durations and bitrates
Media Streaming Benchmark

- Real-Time Streaming Protocol (RTSP)
  - Popular mobile streaming (e.g., m.youtube.com)

Media Streaming Benchmark

- Server required to support RTSP
  - Using Apple Darwin Streaming Server
- Dataset consists of a mix of pre-encoded videos
  - Ten durations: [1 – 10 minutes]
  - Five bitrates: [42 – 1500 kbps]
CloudSuite 1.0

- Data Analytics
- Data Serving
- Media Streaming
- SW Testing
- Web Search
- Web Serving

Software Testing

- Clouds allow dynamic resource allocation as needed
  - Enables previously impossible engineering practices

- Software Testing as a Service leverages cloud env.
  - Large-scale symbolic execution for SW testing
  - Needed as SW scales & complexity increases

- Scale-out engineering application running in cloud
Software Testing Benchmark

- Cloud9, SW Testing as a Service
- Master coordinates symbolic execution
- State maintained in slave, updated from master
- Master load-balances across slaves

CloudSuite 1.0

- Data Analytics
- Data Serving
- Media Streaming
- SW Testing
- Web Search
- Web Serving
Web Search

- Most popular online service
  - Numerous search clouds deployed by industry

<table>
<thead>
<tr>
<th>Query Term</th>
<th>Document</th>
</tr>
</thead>
<tbody>
<tr>
<td>Benchmark</td>
<td>Sample...</td>
</tr>
<tr>
<td>EPFL</td>
<td>Sample...</td>
</tr>
<tr>
<td>Index</td>
<td>Sample...</td>
</tr>
</tbody>
</table>

Web Search Operation

Search User

Frontend

Index Serving Node (ISN)

Inverted Index
Web Search Operation

Search User

Query = “EPFL”

frontend

Caption (2)

ISN

Inverted Index

caption(10)

• Uses Faban traffic generator
• Flexible request mixes
  - # terms /request from published surveys
  - Terms extracted from the crawled dataset

Web Search Benchmark

Search User

frontend

ISN

Inverted Index
Web Search Benchmark

Search User

Frontend

Inverted Index

• Apache Nutch for Front-end & ISNs

ISN

Dataset: Inverted index & captions at ISN
- Generated by crawling public web
- Data at ISN must be memory resident
- Dataset size dictates the number of ISNs
CloudSuite 1.0

- Data Analytics
- Data Serving
- Media Streaming
- SW Testing
- Web Search
- Web Serving

Web Serving

- Key to all internet-based services
- All services are accessed through web servers
- Various technologies construct web applications
  - HTML, PHP, JavaScript, Ruby
Web Serving Operation

Client <-> Web Server <-> Database Server

GET() POST() Query

Web Serving Benchmark

Client Emulator <-> Web Server <-> Database Server

- Faban traffic generator
- Pre-configured page transition matrix (CloudStone)
Web Serving Benchmark

- Web server (Nginx)
- Application server (PHP)
  - Serves a social calendar application (Olio)
- File store (image files)

Web Serving Benchmark

- Database server (MySQL)
CloudSuite: Hands-on

- Media Streaming
  - Installing the server
  - Installing client generator
  - Overview of the dataset
  - Running the benchmark
  - Checking quality of service

CloudSuite
Full-System Simulation

Cansu Kaynak
CloudSuite Simulation Requirements

CloudSuite Workloads:
- Multi-threaded, multi-processor
- Data-intensive
- Multi-tier

⇒ Exercise OS and I/O extensively
⇒ OS and I/O are first-order performance determinants

Need full-system simulation

Flexus Framework

- Functional Full-System Simulation: Simics
- Detailed Microarchitectural Simulation: Flexus
- Fast Simulation: Statistical sampling
Flexus Framework

- **Functional Full-System Simulation:** Simics
- **Detailed Microarchitectural Simulation:** Flexus
- **Fast Simulation:** Statistical sampling

Full-System Simulation Requirements

Full-system functional simulator must support:

- Privileged-mode ISA
- I/O devices
- Networks of systems
- Saving/restoring architecturally-visible state

**Simics provides these capabilities**
Simics Configuration & CLI

- Configuration file defines system components
  - Motherboard, CPUs, memory, I/O devices

- Command-line interface (CLI) provides interface to simulation
  - Start and stop simulation
  - Save and restore target system checkpoints

Simics Checkpoints

- Contain full-system architectural state
- Are incremental - Require all files in chain
- Form the basis for Flexus simulation
Simics μArch Interface

- Simics does not provide timing details
  - But provides a Micro-Architectural Interface (MAI)
  - Allows a user module to take control over timing
- Simics feeds Flexus with instructions
- Flexus gives timing feedback to Simics

Simics Hands-On
Preparing a Workload for Simulation

1. Install OS
2. Booting target machine
3. Install application & create dataset
4. Tune workload parameters
5. Run application
Media Streaming in Simics Hands-on

1. Loading a freshly-installed OS checkpoint
2. Preparing target system
3. Running applications in Simics
4. Saving system checkpoints
5. Loading system checkpoints

Initial Checkpoint

- Freshly-installed OS: Solaris 10 u9
- Media Streaming binaries & datasets
  - Faban client on Client machine
  - Darwin Streaming Server on Server machine
  - Video dataset on Server machine
- Necessary libraries
Getting Started with Media Streaming

Simulated target system:
- Server (1 core)
- Client (1 core)

- Binaries:
  /opt
- Dataset:
  /streaming_data

Preparing Target System

- Move configuration files
- Move experiment files
- Start experiment
Media Streaming in Action

- Monitoring
- QoS check

Flexus Simulator Toolset

Stavros Volos
Software Simulation

- Allows for fast & easy evaluation of an idea
  - Minimal cost, simulator runs on your desktop
  - Reuse components, don’t implement everything

- Enables various benchmarks (e.g., SPEC, CloudSuite)
  - 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
- Simulators and components in Flexus 4.1
- Hands-on

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
- Simulators and components in Flexus 4.1
- Hands-on

Flexus organization

- /components
  - Cache
  - Interconnect
  - Feeder
- /simulators
  - CMP.OoO
  - UP.OoO
- /core
  - Debug
  - Stats
  - Simics Interface

FLEXUS_ROOT
**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

```c
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**

```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) { ... }
```

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**
  - Simulators and components in Flexus 4.1
  - Hands-on
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:
  ```cpp
  Stat::StatCounter myCounter(statName() + "-count");
  ++ myCounter;
  ```
A typical debug statement

```cpp
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-interface)
- 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

- Simulators and components in Flexus 4.1
- Hands-on
Simulators in Flexus 4.1

- UP.Trace
- CMP.L2Shared.Trace
- CMP.MT4.L2Shared.Trace
- fast memory system
- fast CMP memory system
- fast CMP memory system w/ 4-way MT support
- UP.OoO
- CMP.L2SharedNUCA.OoO
- CMP.MT4.L2SharedNUCA.OoO
- private L1 / shared L2
- private L1 / shared L2 w/ 4-way MT support
- CMP.L2SharedNUCA.DRAMSim.OoO
- private L1 / shared L2 w/ DRAMSim 2.0

Memory hierarchy

- “top”, “front” = closer to CPU
- Allows 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’02]
  - 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 consistency models (SC, TSO, RMO)

Hands-on

- Set 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?
### Flexus: The Way to Full-System Cycle-Accurate Simulation

<table>
<thead>
<tr>
<th>Simulator</th>
<th>Full-system Simulation Support</th>
<th>Simulation Speeds</th>
<th>Statistical Sampling Support</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Fast (MIPS)</td>
<td>Trace (KIPS)</td>
</tr>
<tr>
<td>Flexus</td>
<td>✓</td>
<td>30-60</td>
<td>750-850</td>
</tr>
<tr>
<td>gem5</td>
<td>✓</td>
<td>3-5</td>
<td>35-350</td>
</tr>
<tr>
<td>ISA-only</td>
<td>×</td>
<td>Do not support full-system simulation</td>
<td></td>
</tr>
</tbody>
</table>

### Boosting Simulation Speed with Statistical Sampling

Pejman Lotfi-Kamran
Simulation Speed Challenges

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

- Slower simulators
  - Full-system simulation: 1000× slower than SimpleScalar

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

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

Full-system simulation is slow

- Simulation slowdown per cpu
  - Real HW: ~ 2 GIPS 1 s
  - Simics: ~ 30 MIPS 66 s
  - Flexus, no timing: ~ 900 KIPS 37 m
  - Flexus, OoO: ~ 24 KIPS 23 h

2 years to simulate 10 seconds of a 64-core workload!
Statistical Sampling

- Random selection of population
  - E.g., 3000 out of 300 million
- Predict the behavior based on the selected sample

- Features:
  - High accuracy
  - Simple
  - Strong mathematical foundation

Power of a small part to predict behavior of a whole

Statistical Sampling for Simulation

- Measure uniform or random locations

- Each measurement is on a group of instructions

- ~10,000x reduction in turnaround time

Challenge: programs are sequential
Sampling of Sequential Programs

- Correctness
  - State of memory, registers, etc.
- Bias
  - State of cache, branch predictor, reorder buffer, etc.

Functional Simulation

- Functional simulation is faster than detailed simulation
  - Flexus (no timing) is 38 times faster than Flexus (OoO)
- Use functional simulation for “warmup”
  - Memory (guarantees correctness)
  - Cache hierarchy (avoids bias)
  - Branch predictor (avoids bias)
Handling Bias

• Core micro-architecture can be warmed up rapidly
  – Detailed simulation to warmup core micro-architecture

• Perform warmup prior to measurement
  – Functional warming during fast-forwarding
  – Detailed warmup before each simulation window

SMARTS

Functional warming  Detailed warmup  Measurement

Simulation Speedup

• 10 seconds of a 64-core workload
  – Normal execution: 2 years
  – With sampling: 20 days

• 37x improvement in simulation speed but not enough
• Solution
  – Avoid functional simulation (17 days)
  – Accelerate detailed simulation (3 days)
Avoiding Functional Simulation

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

*Works for any microarchitecture*

Accelerating Detailed Simulation

- Checkpoint library makes measurement independent
- Run multiple measurements in parallel

Run in parallel
**Simulation Speedup**

- Sampling without a checkpoint library:
  - 10 seconds of a 64-core workload: 20 days

- Sampling with a checkpoint library:
  - 10 seconds of a 64-core workload: 3 hours with 100 cores

**How to Choose the Sample Size?**

- Low variability ➔ Small sample size
- High variability ➔ Large sample size

*Variability determines sample size*
Steps for Timing Simulation

1. Prepare workload for simulation
   – Port workload into Simics

2. Measure baseline variance
   – Determine required library size

3. Collect checkpoints
   – Via functional warmup

4. Detailed Simulation
   – Estimate performance results

2. Determine Sampling Parameters

- Guess variability
- Generate flexpoints for the variability
- Run timing simulation
- Measure error and correct the guess
Typical Sampling Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Flexus (64-CPU CMP.OoO)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Warming</td>
<td>100k cycles</td>
</tr>
<tr>
<td>Measurement</td>
<td>50k cycles</td>
</tr>
<tr>
<td>Target confidence</td>
<td>95%</td>
</tr>
<tr>
<td>Sample size</td>
<td>800</td>
</tr>
<tr>
<td>Sim. time per checkpoint</td>
<td>~ 20 min</td>
</tr>
</tbody>
</table>

3. Checkpoint Creation

- Spread Simics checkpoints
  - Simics fast mode rapidly covers 10 seconds

- Collect flexpoints in parallel
  - Via CMP.L2Shared.Trace
  - From each Simics checkpoint

© 2006-2012 A. Adileh, M. Ferdman, C. Kaynak, P. Lotfi-Kamran, S. Volos, T. Wenisch, R. Wunderlich
4. Detailed Simulation

- Run detailed simulation with OoO simulators
- Process all flexpoints, aggregate offline
- Manipulate results 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 delta reduces sample size by 3.5 to 150x

Matched-pair with Flexus

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

- Complex changes (e.g., adding components)
Hands-on

- Generate Flexpoints
- Launch timing simulation for all flexpoints
- Aggregate stats with stat-collapse
- Examine aggregate statistics
  - Compute confidence
  - Plot timing breakdown

How to Use CloudSuite Images

Cansu Kaynak
CloudSuite Simics Release

Released images (phase_000) contain:
- CloudSuite binaries & necessary libraries
- Tuned workloads at steady state
- Ready to run

CloudSuite Images

*From 1 core to 64 cores:*
1. Data Analytics
2. Data Serving
3. Media Streaming (4, 8, 16 cores)
4. Software Testing
5. Web Search (1 to 32 cores) ~ SW scalability
6. Web Serving (1 to 8 cores)
Deploying CloudSuite Images

- Paths for logical components in configuration files:
  - Binary disk
  - Data disk(s)

```python
checkpoint_path: ( "/path/to/binary_disk", "/path/to/data_disk")
```

- Load initial state & save it as phase_000
- Detailed instruction are in setup document...

Directory Hierarchy for Flexus

```
Workload
  1cpu   2cpu   ...   Ncpu
    |      |      |      |
    |      |      |      |
    baseline
      |      |      |      |
      |      |      |      |
      phase_000   ...   phase_N
          |      |      |
          |      |      |
          |      |      |
          flexpoint_001   ...   flexpoint_M
              |      |      |
              |      |      |
              user_name   simics
                  |      |
                  |      |
                  |      |
                  |      |
                  fxpt_name
```
**What We Release**

We provide phase_000:
- Steady state of workload execution

**How Long To Simulate**

Representative execution window of a workload:
- Steady architectural behavior (measured on real HW)
- 10 sec. of native execution (25 sec. for media streaming)
Phase Generation

Divides the entire execution into phases
• Generates phases (Simics checkpoints) using Simics fast mode
• As many phases as necessary for desired parallelism
  – e.g., 10 phases

Flexpoint Generation

Divides every phase into flexpoints (parallel across phases)
• Generates flexpoints using Flexus trace simulator
  – Functional warming of cache and branch predictor state
• As many flexpoints as necessary for desired degree of confidence
  – e.g., 80 flexpoints per phase
Timing Simulation

Cycle-accurate simulation in parallel across flexpoints
- First, detailed warm-up of microarchitectural state
- Then, takes measurements from the warmed state
  - e.g., 100K-cycle warm-up, 50K-cycle measurement
  - Longer warm-up necessary for Data Serving

Independent parallel simulations

Execution

Wrap-Up

- Two steps before cycle-accurate simulation:
  1. Phase generation
  2. Flexpoint generation

- Refer to .run_job.rc.tcl in Flexus 4.1 for workloads, phases, flex-points
Thanks!