Modeling and Parallel Simulation of Multicore Architectures with Manifold

The Manifold Team

School of Electrical and Computer Engineering and School of Computer Science
Georgia Institute of Technology
Atlanta, GA. 30332

Oracle Labs

Sponsors

Motivation

“Remember that all models are wrong; the practical question is how wrong do they have to be to not be useful.”

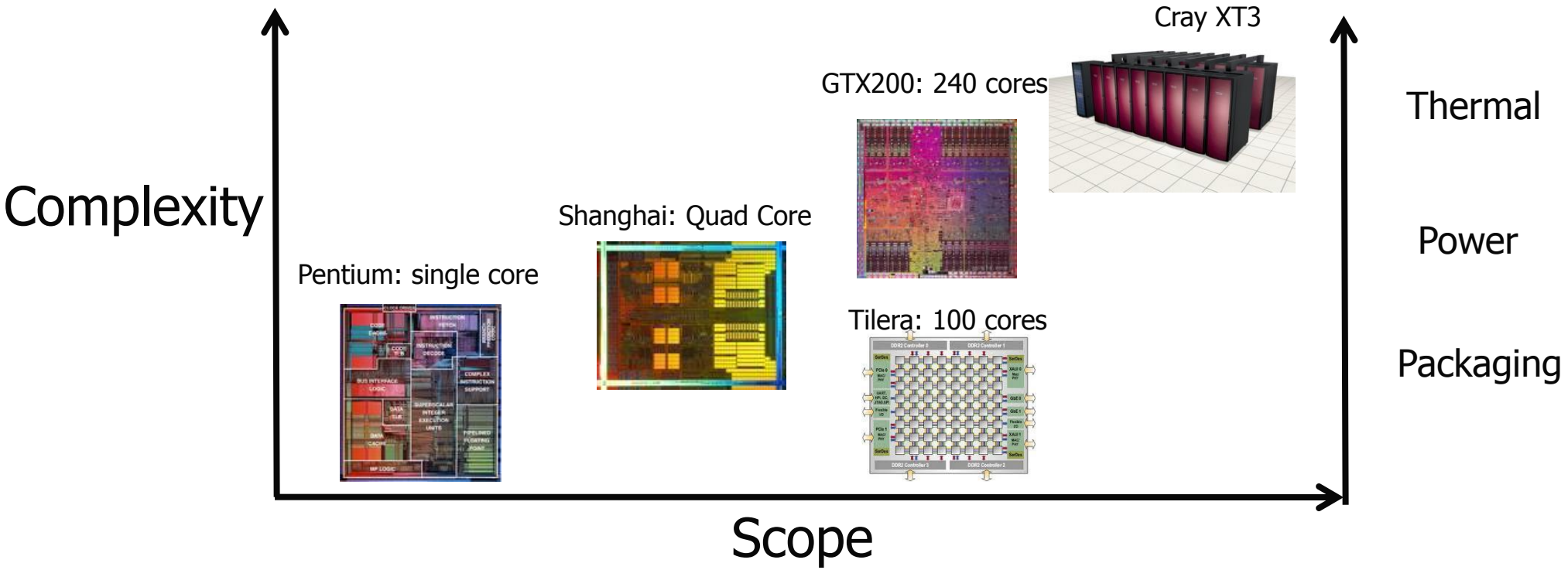
Box, G. E. P., and Draper, N. R., (1987), *Empirical Model Building and Response Surfaces*, John Wiley & Sons, New York, NY.

George E. P. Box, 2011

Manifold@GT

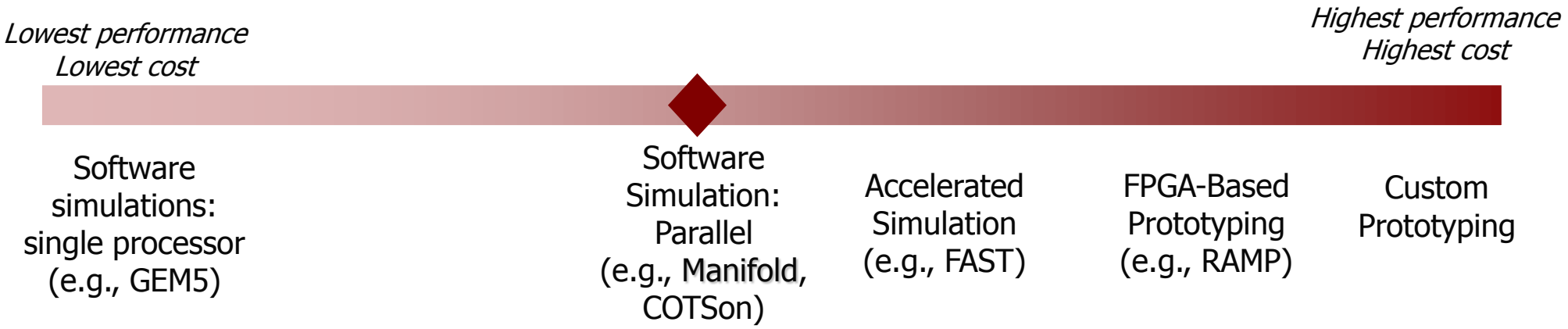
- Faculty
  - Tom Conte (SCS)
  - George Riley (ECE)
  - S. Yalamanchili (ECE)
- Research Staff
  - Jun Wang (ECE)
- Collaborators
  - Genie Hsieh (Sandia)
  - Saibal Mukhopadhyay (ECE)
  - Hyesoon Kim (SCS)
  - Arun Rodrigues (Sandia)
- Graduate Students
  - Jesse Beu
  - Rishiraj Bheda
  - Zhenjiang Dong
  - Chad Kersey
  - Elizabeth Lynch
  - Jason Poovey
  - Mitchelle Rasquinha
  - William Song
  - He Xiao
  - Peng Xu
  - …+ many other contributors

Modeling and Simulation Demands



- System complexity is outpacing simulation capacity
  - Cannot perform analysis at scale
- The problem is getting worse faster → Simulation Wall
- Today - islands of simulators and simulation systems
  - Customized interactions
  - Difficult to leverage individual investments

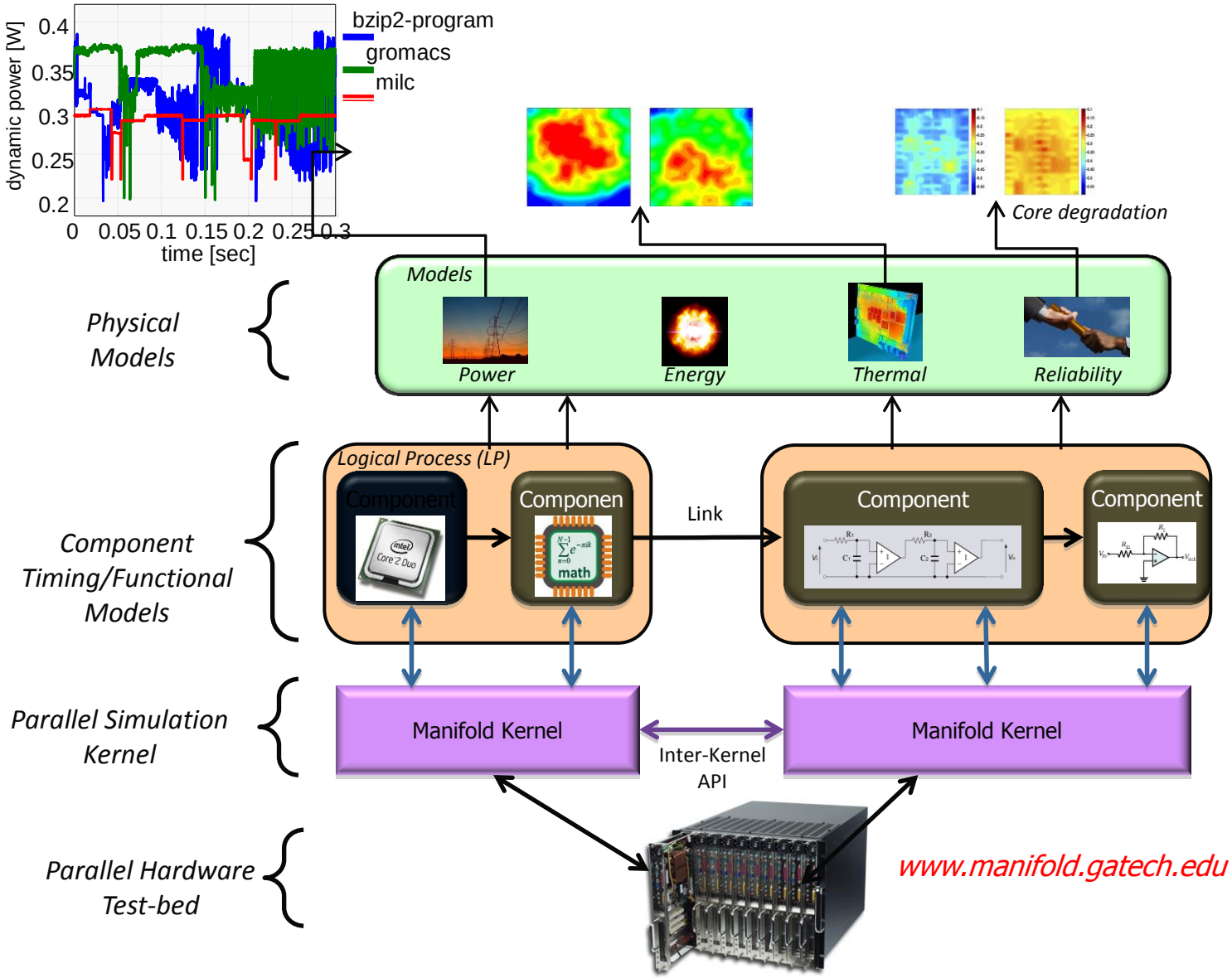
Spectrum of Solutions



- Simple Premise: Use parallel machines to simulate/emulate parallel machines
- Leverage mature point tools via standardized API for common services
  - Event management, time management, synchronization
  - Learn from the PDES community
- Cull the design space prior to committing to hardware prototyping or hardware acceleration strategies

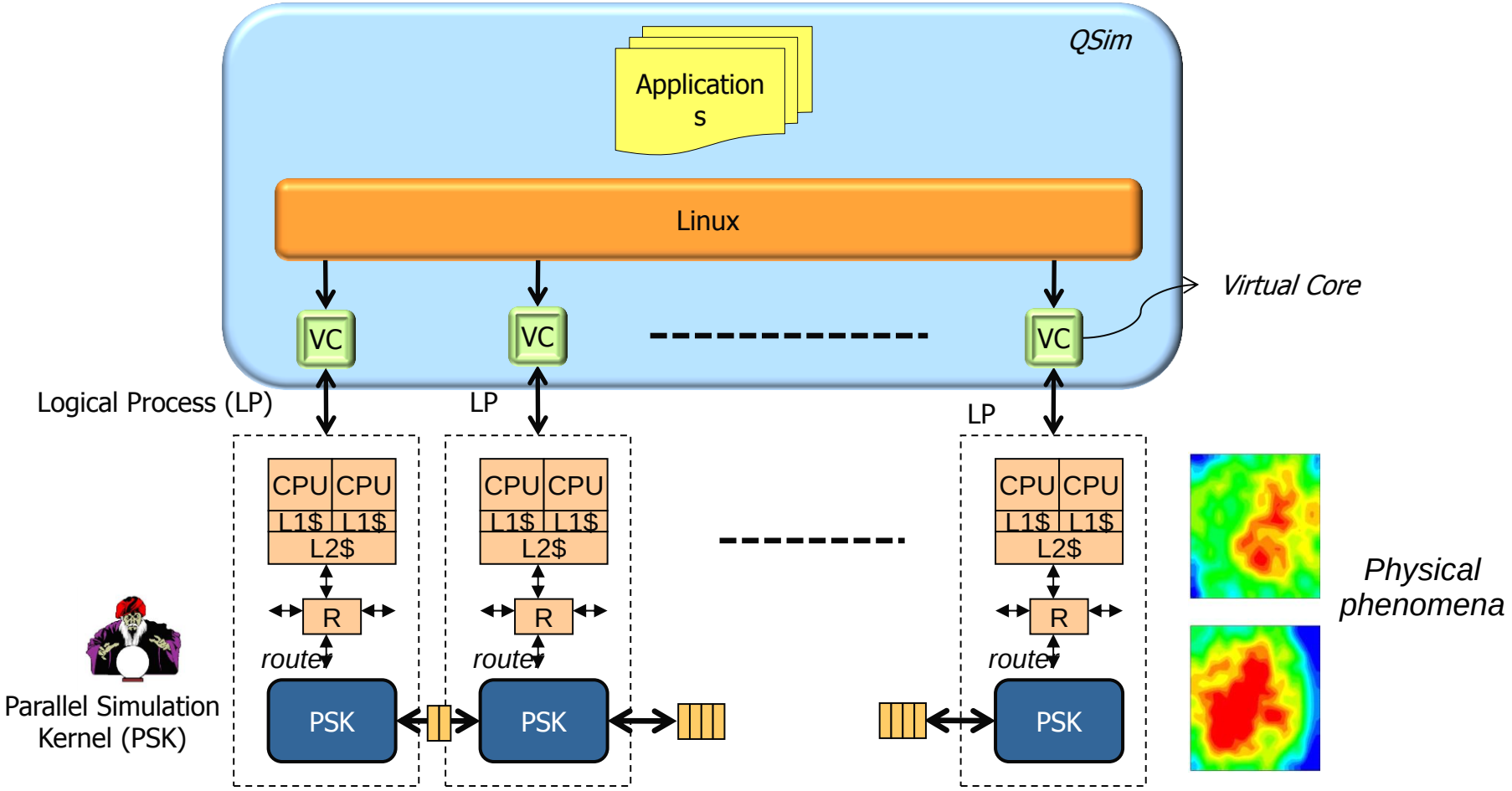
Manifold: The Big Picture

A composable parallel simulation system for heterogeneous, many core systems.



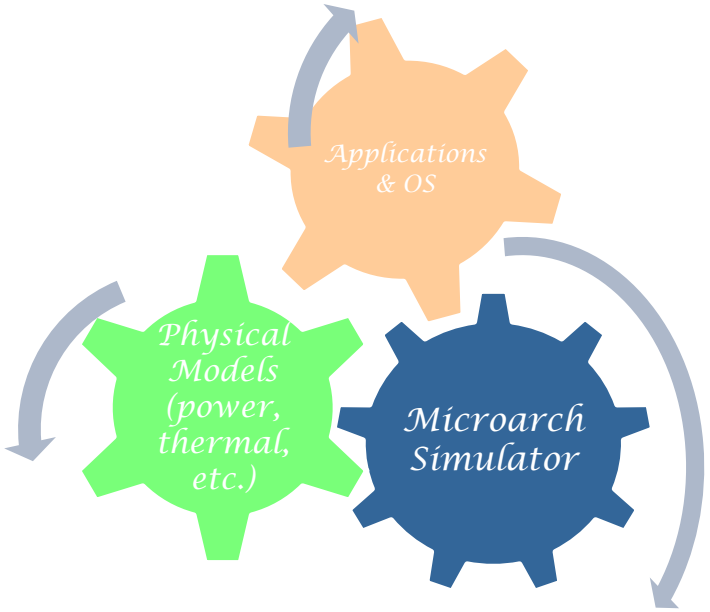
- Component-based and extensible
- Mixed discrete event and time stepped simulation
- From full system HW/SW models to abstract timing models
- From detailed cycle-level to high level analytic models
- Integration of third party tools

A Typical Single OS Domain Model



- Single Socket/Board model → scaling across multiple sockets
- Similar efforts: Graphite, GEM5/SST, Sniper, etc.

Simulation Infrastructure Challenges

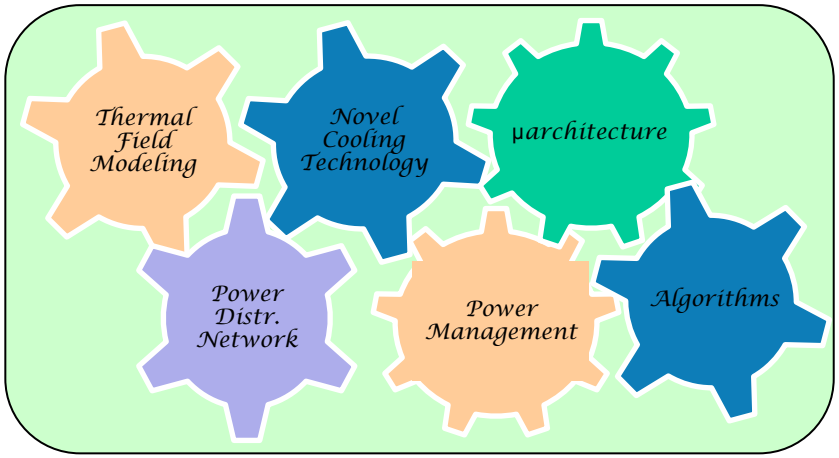


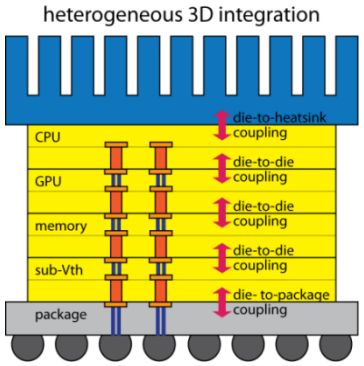
- Scalability
  - Processors are parallel and tools are not → not sustainable
- Multi-disciplinary
  - Functional + Timing + Physical models
- Need to model complete systems
  - Cores, networks, memories, software at scale
- Islands of expertise
  - Ability to integrate point tools → best of breed models
- Composability
  - Easily construct the simulator you need

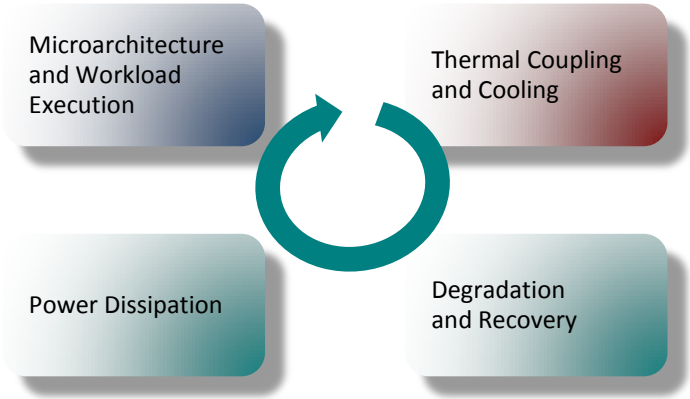
Goal

Not to provide a simulator,
but
Make it easy to construct a validated simulator at the fidelity and scale you want, and

Provide base library of components to build useful multicore simulators







Tutorial Schedule

	Topical Description
15 minutes	Introduction and Overview
30 minutes	Execution Model and Software Organization
90 min	Component Models
15 minutes	Break
30 minutes	Building and Running Simulations
45 minutes	Energy Introspector: Integration of Physical Models
15 minutes	Some Example Simulators

Outline

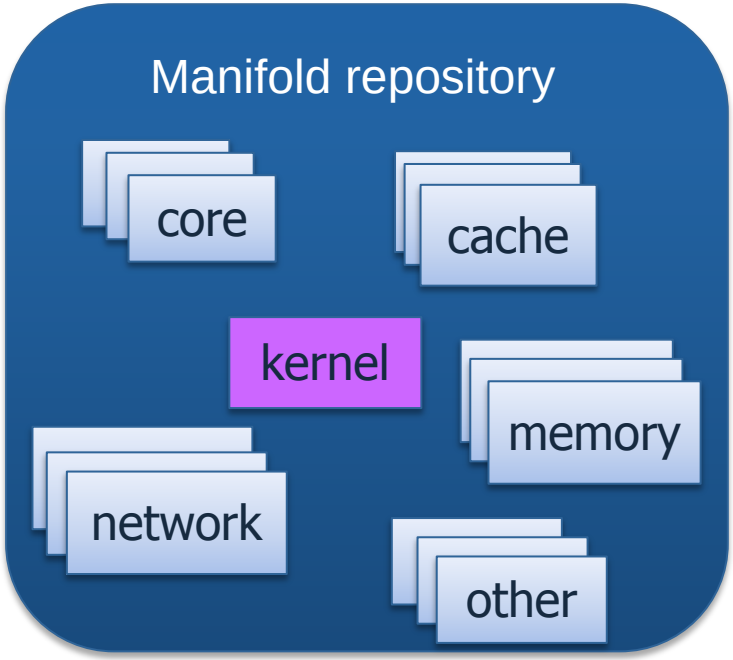
- Introduction
- Execution Model and System Architecture
- Multicore Emulator Front-End
- Component Models
  - Cores
  - Network
  - Memory System
- Building and Running Manifold Simulations
- Physical Modeling: Energy Introspector
- Some Example Simulators

Manifold Execution Model and System Architecture

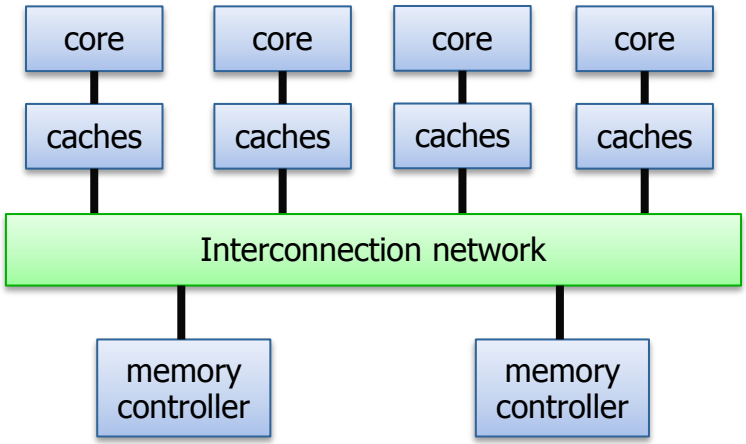
- Execution model
- Software architecture
- Simulation kernel
- Manifold component
- Building system models

Manifold Overview

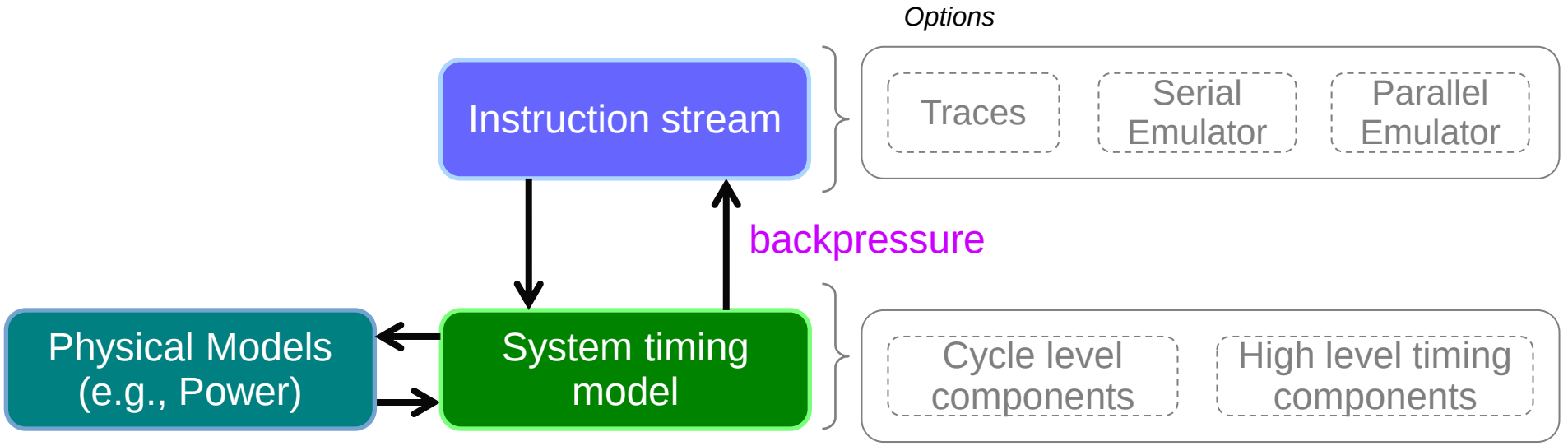
- A parallel simulation framework for multicore architectures
- Consists of:
  - A parallel simulation kernel
  - A (growing) set of architectural components
  - Integration of physical models for energy, thermal, power, and so on
- Goal: easy construction of parallel simulators of multicore architectures



Ease of composition

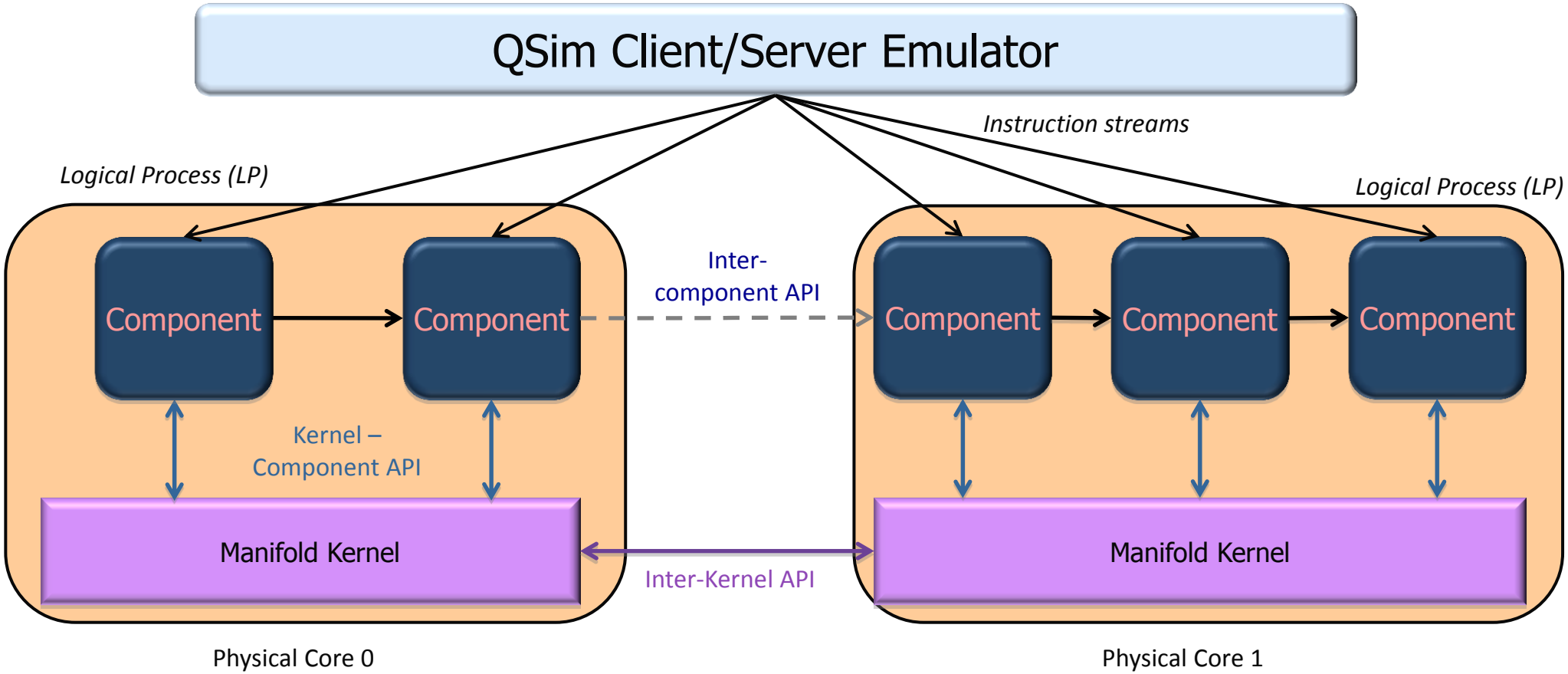


Execution Model: Overview



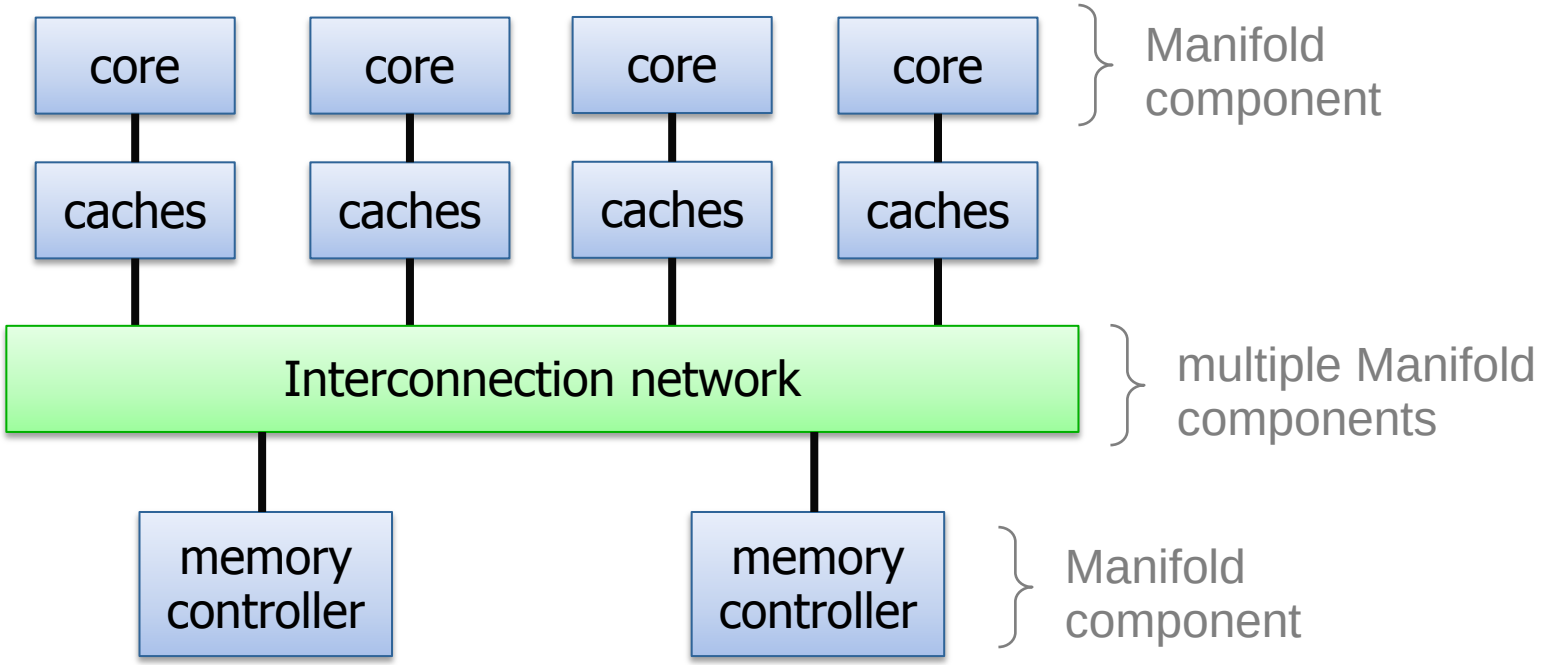
- Instruction stream
  - Generated by i) trace files, ii) Qsim server, iii) Qsim Lib
- System timing model
  - Multicore model built with Manifold components
  - Components assigned to multiple logical processes (LPs)
    - Each LP assigned to one MPI task; LPs run in parallel

Execution Model: Software Organization



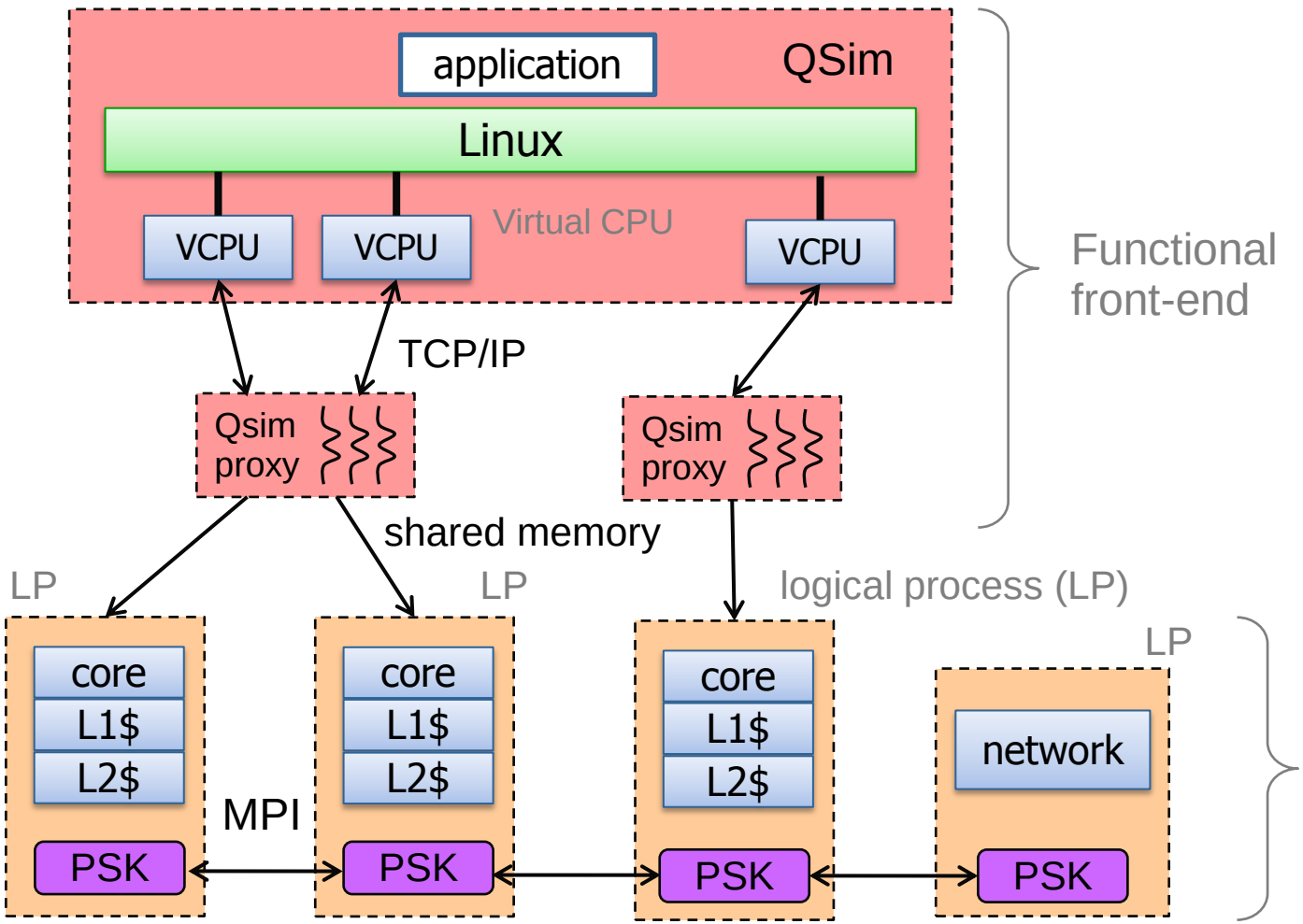
- Component models connected to form system models
- Full system emulation front-end
- All time management is handled by the simulation kernel
- APIs → key to integrating mature point tools

An Example System Model



- Multicore SMP system
- Core, cache, memory controller models each represented by a Manifold component
- Network contains multiple components: interfaces, routers

Execution Model



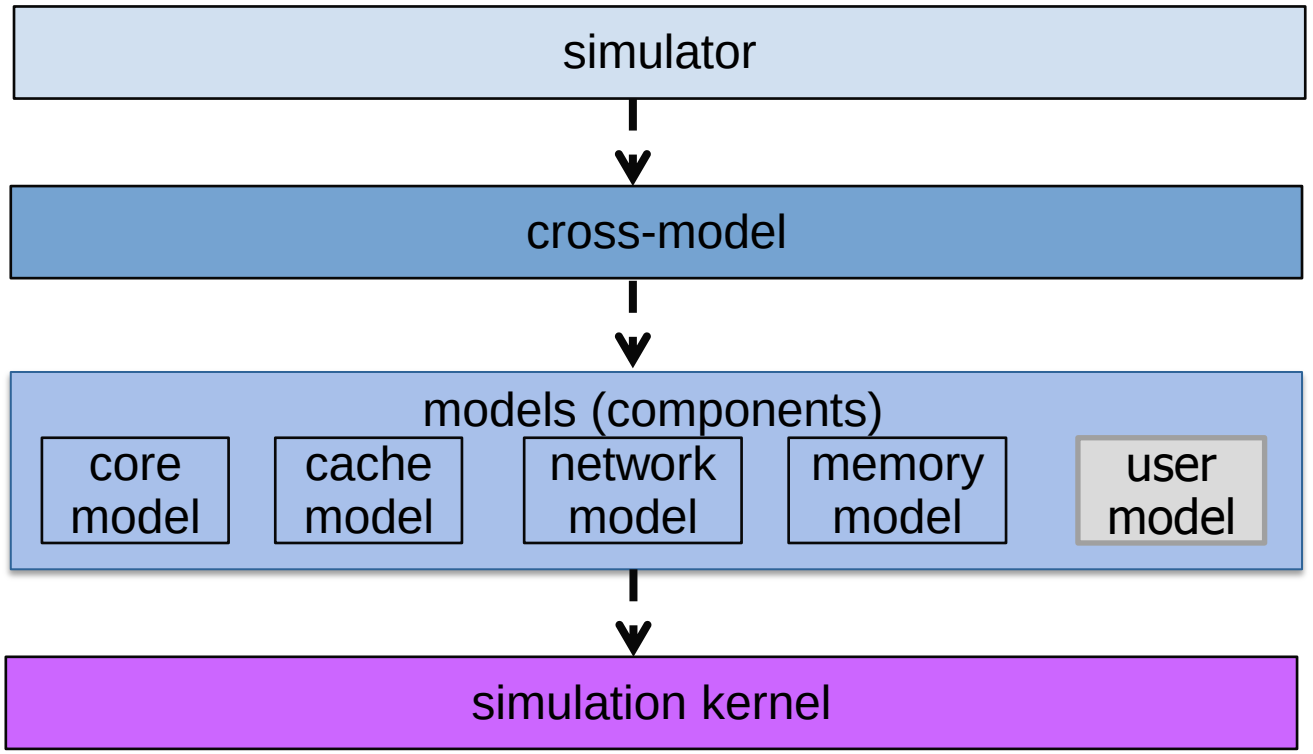
- Timing-directed full-system simulation
- Front-end performs functional emulation; timeless
- Back-end for timing simulation
- Front-end is regulated by back-end

Manifold Execution Model and System Architecture

- Execution model
- Software architecture
- Simulation kernel
- Manifold component
- Building system models

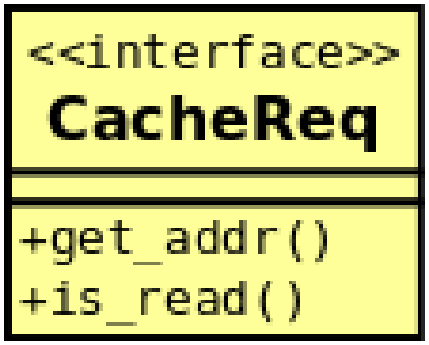
Software Architecture (1/2)

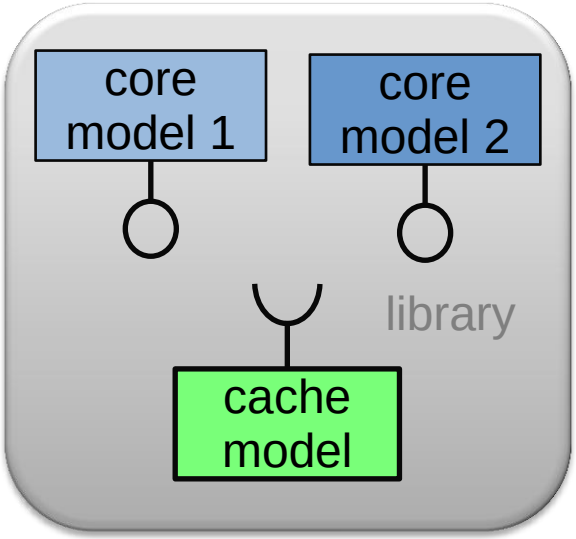
- Layered architecture
- Standardized component interfaces
- Goal:
  - Encapsulation of parallel-discrete event simulation functionality
  - Allow components to evolve independently
  - plug-n-play construction of simulators

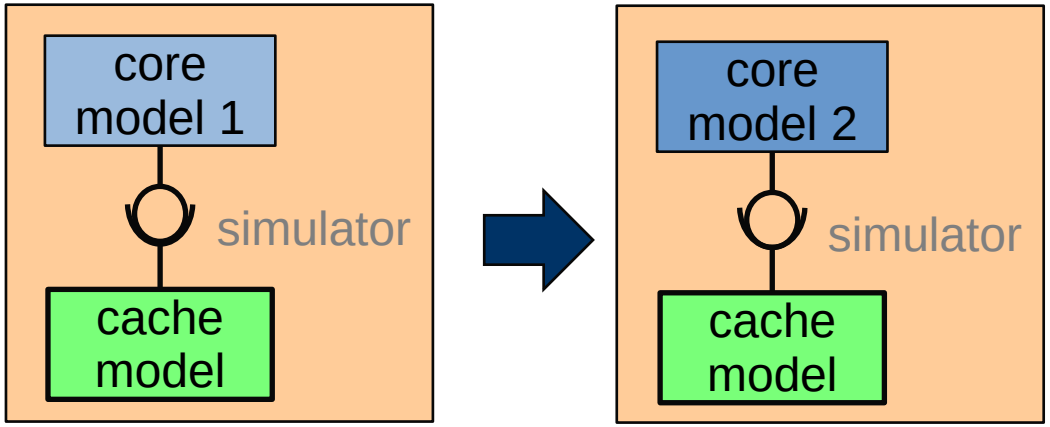


Software Architecture (2/2)

- Standardized component interfaces
  - allows components to be independent of each other: no compile-time dependence
  - allows components to be interchangeable
- Example
  - processor-cache interface







Source Code Organization

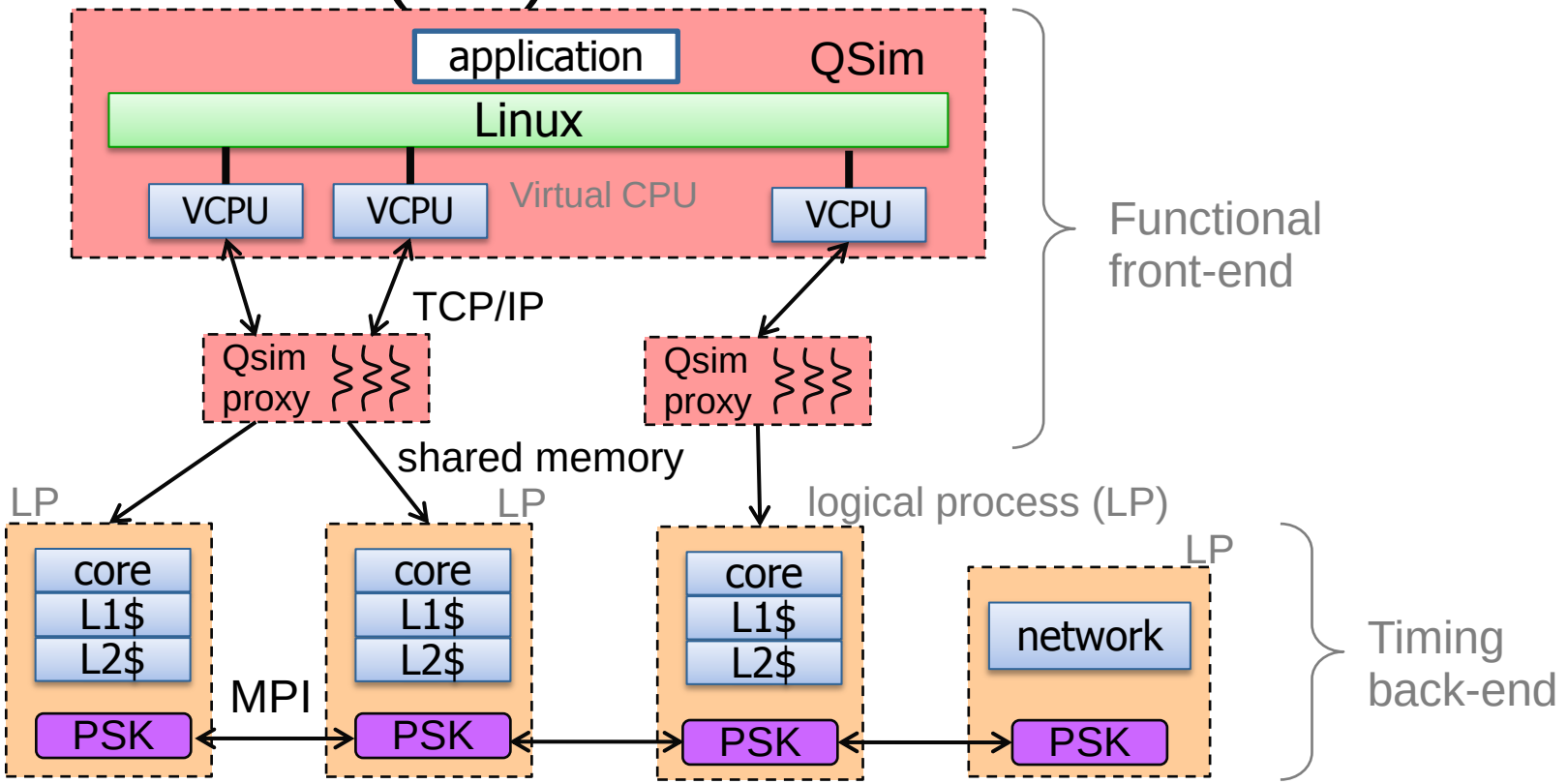
```
code
|... doc
|... kernel
|... models
|    |... cache
|    |... cross
|    |... energy_introspector
|    |... memory
|    |... network
|    |... processor
|... simulator
|... uarch (common micro-architecture classes)
|... util (utility programs)
```

- How to get the code?
  - Distribution package: http://manifold.gatech.edu/download
  - SVN: https://svn.ece.gatech.edu/repos/Manifold/trunk

Manifold Execution Model and System Architecture

- Execution model
- Software architecture
- Simulation kernel
- Manifold component
- Building system models

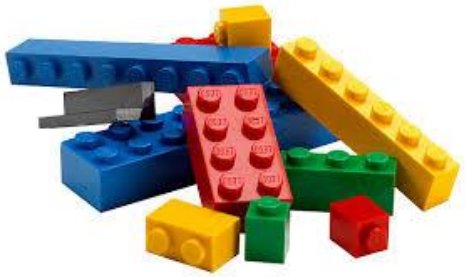
Simulation Kernel (1/2)



- Simulation Kernel
  - provides facilities for creating / connecting components
  - provides clock related functionalities
  - Encapsulates parallel discrete-event simulation (PDES) services
    - Transparent synchronization between parallel LPs
    - All event management

Simulation Kernel (2/2)

- Interface
  - Component functions
    - create / connect components
    - send output
  - Clock functions
    - create clocks
    - register component with clock
    - Support for dynamic voltage frequency scaling (DVFS)
  - Simulation functions
    - start / stop simulation
    - statistics



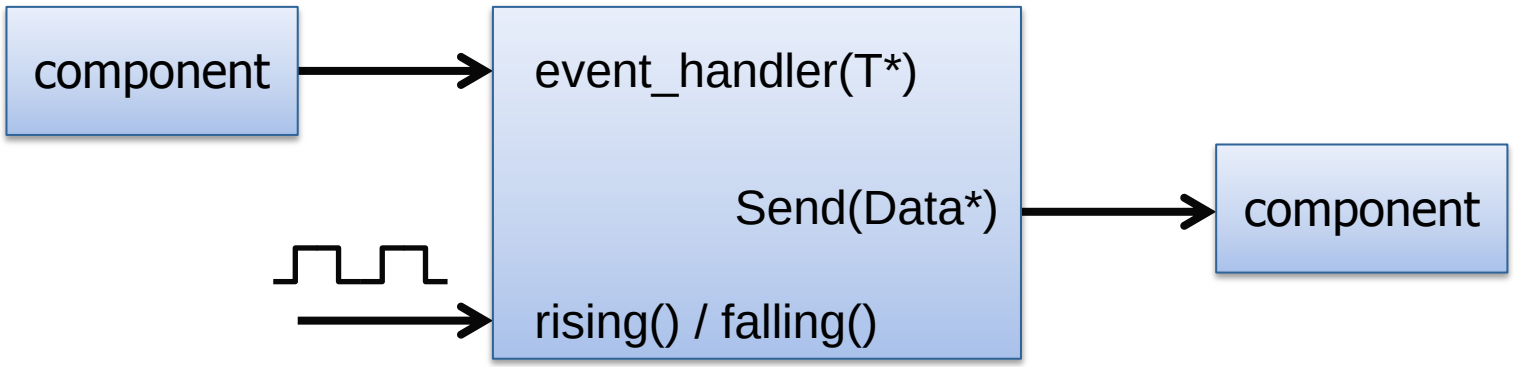
- Encapsulated part
  - PDES engine: event management, event handling, inter-process communication, synchronization algorithms

Manifold Execution Model and System Architecture

- Execution model
- Software architecture
- Simulation kernel
- Manifold component
- Building system models

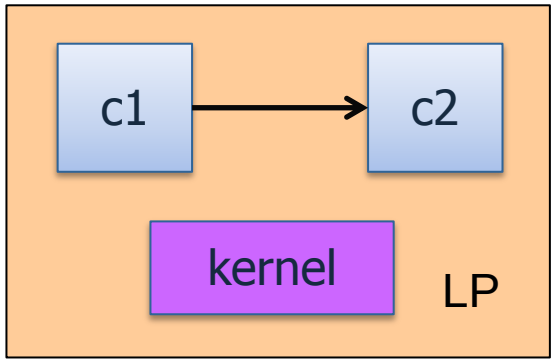
Operational Model of a Component (1/3)

- Component is connected to other components via links
- For each input port, there should be an event handler
- Can register one or two functions with a clock; registered functions are called every clock cycle
- For output, use Send()
  - Paradigm shift: OO →event-driven
    - Instead of comp->function(data), call Send(data)
    - No method invocation: comp->function()
    - May not even have a valid pointer: component could be in another process
  - Kernel ensures receiver's handler is called at the right moment



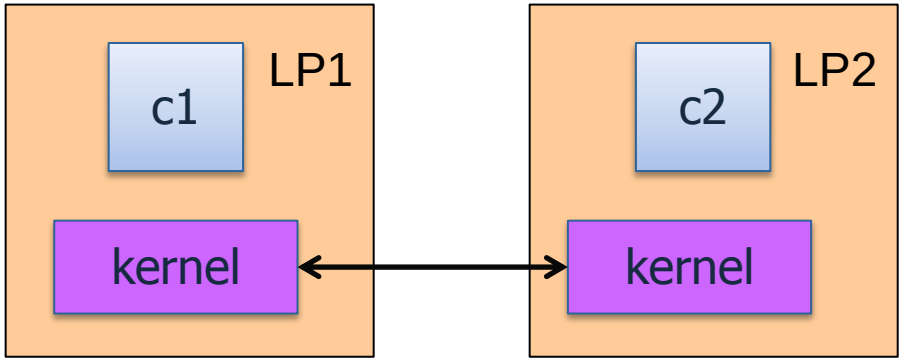
Operational Model of a Component (2/3)

- How is the event handler of component invoked?
- Example: component c1 sends data at cycle t to component c2; link delay is d cycles
  - What's expected: c2's handler is called at cycle (t+d)
- **Case 1: c1 and c2 in same LP**
  - When c1 is connected to c2, a LinkOutput object is created that holds a pointer to c2, and a pointer to its handler.
  - When c1's Send() is called, the LinkOutput object calls kernel's schedule function to schedule an event that calls c2's handler at (t+d).



Operational Model of a Component (2/3)

- **Case 2: c1 and c2 in different LPs**
  - When c1 is connected to c2, a LinkInput object for c2 is created that holds a pointer to its handler.
  - When c1's Send() is called, kernel1 sends data to kernel2, which passes it to c2's LinkInput object, which calls kernel2's schedule function to schedule an event that calls c2's handler at (t+d).



- The same Send() function is used in both cases!

Manifold Component (1/2)

- Must be a subclass of manifold::kernel::Component
- Must define event handlers for incoming data
- Must use Send() to send outputs
- Define functions for rising/falling edge if required
- For more, see *Manifold Component Developer's Guide*

```
class MyComponent : public manifold::kernel::Component
{
public:
    enum {PORT0=0, PORT1};
    void handler0(int, MyDataType0*);
    void handler1(int, MyDataType1*);
    void rising();
    void falling();
};
```

Manifold Component (2/2)

- Important functions in manifold::kernel::Component
- Create()
  - create a component
  - 5 overloaded template functions
- Send()
  - send data out of a port to a component connected to the port
  - no recv() function: incoming events handled by event handler.

```
template<typename T, typename T1)
void Create(int, T1, CompName);

template<typename T>
void Send(int, T);
```

Manifold Execution Model and System Architecture

- Execution model
- Software architecture
- Simulation kernel
- Manifold component
- Building system models

Building System Models and Simulation Programs (1/2)

- Steps for building a simulation program
  - Initialization
  - Build system model (see next slide)
  - Set simulation stop time
  - Start simulation
  - Finalization
  - Print out statistics

Building System Models and Simulation Programs (2/2)

- Building a system model
  - Create clock(s) (call constructor)
  - Create components (call Component::Create())
  - Connect components (call Manifold::Connect())
  - Register clock-driven components with clock, if not already registered. (call Clock::Register())

Outline

- Introduction
- Execution Model and System Architecture
- Multicore Emulator Front-End
- Component Models
  - Cores
  - Network
  - Memory System
- Building and Running Manifold Simulations
- Physical Modeling: Energy Introspector
- Some Example Simulators

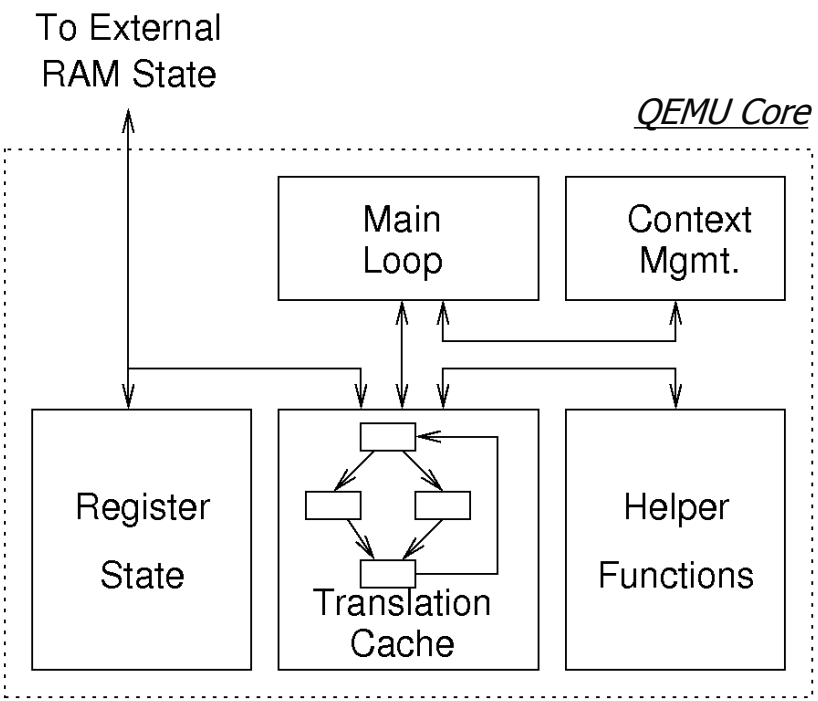
QSim[1]: Overview

- Thread safe multicore x86 emulation library using QEMU[2]
  - C++ API for instantiating, controlling, and instrumenting emulated CPUs
- **Guest** environment runs:
  - Lightly modified Linux kernel
  - Unmodified 32-bit x86 binaries
- Instruction-level execution control
- Instruction-level instrumentation
- Qsimlib : for creating multithreaded emulators
- QSimServer: for serving parallel/distributed simulations

[1]C. Kersey, A. Rodrigues, and S. Yalamanchili, " A Universal Parallel Front-End for Execution-Driven Microarchitecture Simulation," *HIPEAC Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, January 2012

[2]F. Bellard, "QEMU Open Source Processor Emulator," *http://www.qemu.org/*

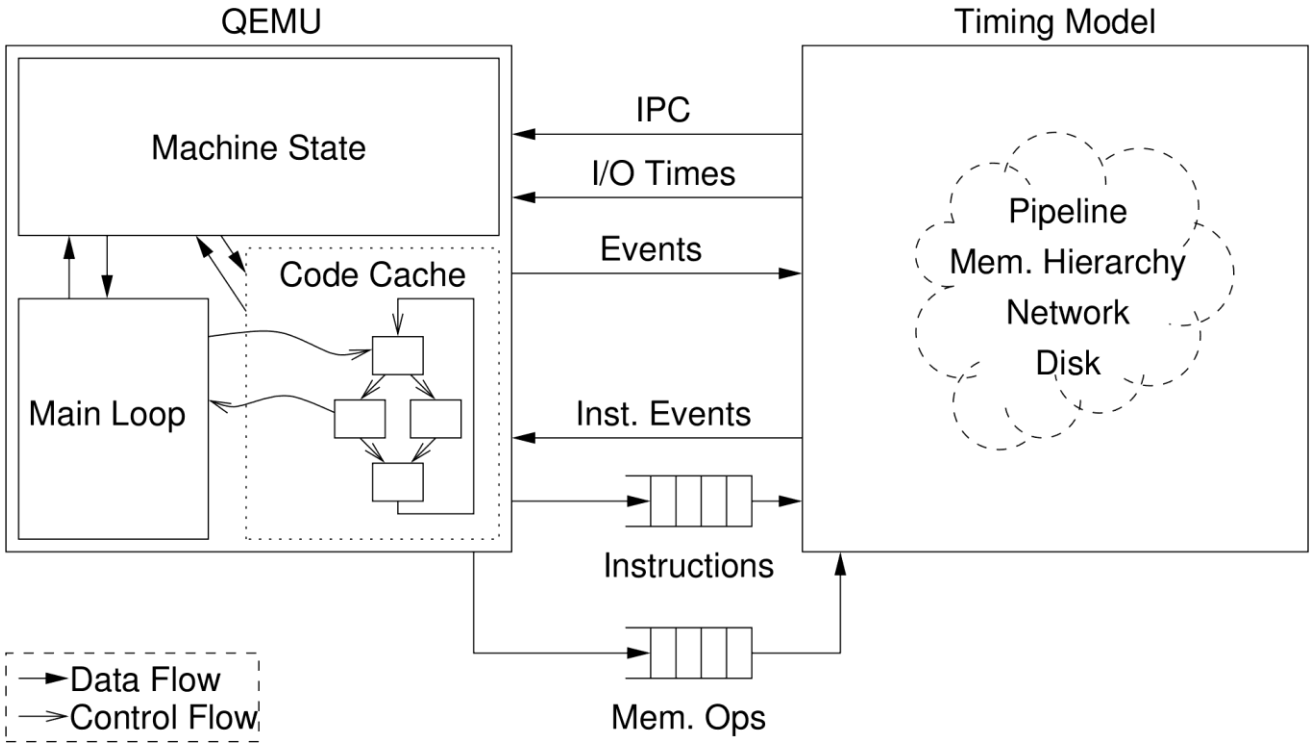
QSim Architecture: QEMU Core



- Emulation performed through dynamic binary translation.
- Code from **translation cache** can call **helper functions**
  - Instrument code cache

- QSim's QEMU CPU library uses external Guest RAM state.
  - Allows sharing between multiple instances
- Synchronization allows multiple ordinary QEMU CPUs or a single QEMU CPU performing an atomic memory operation to run.

QSim Architecture: Timing Model Interface

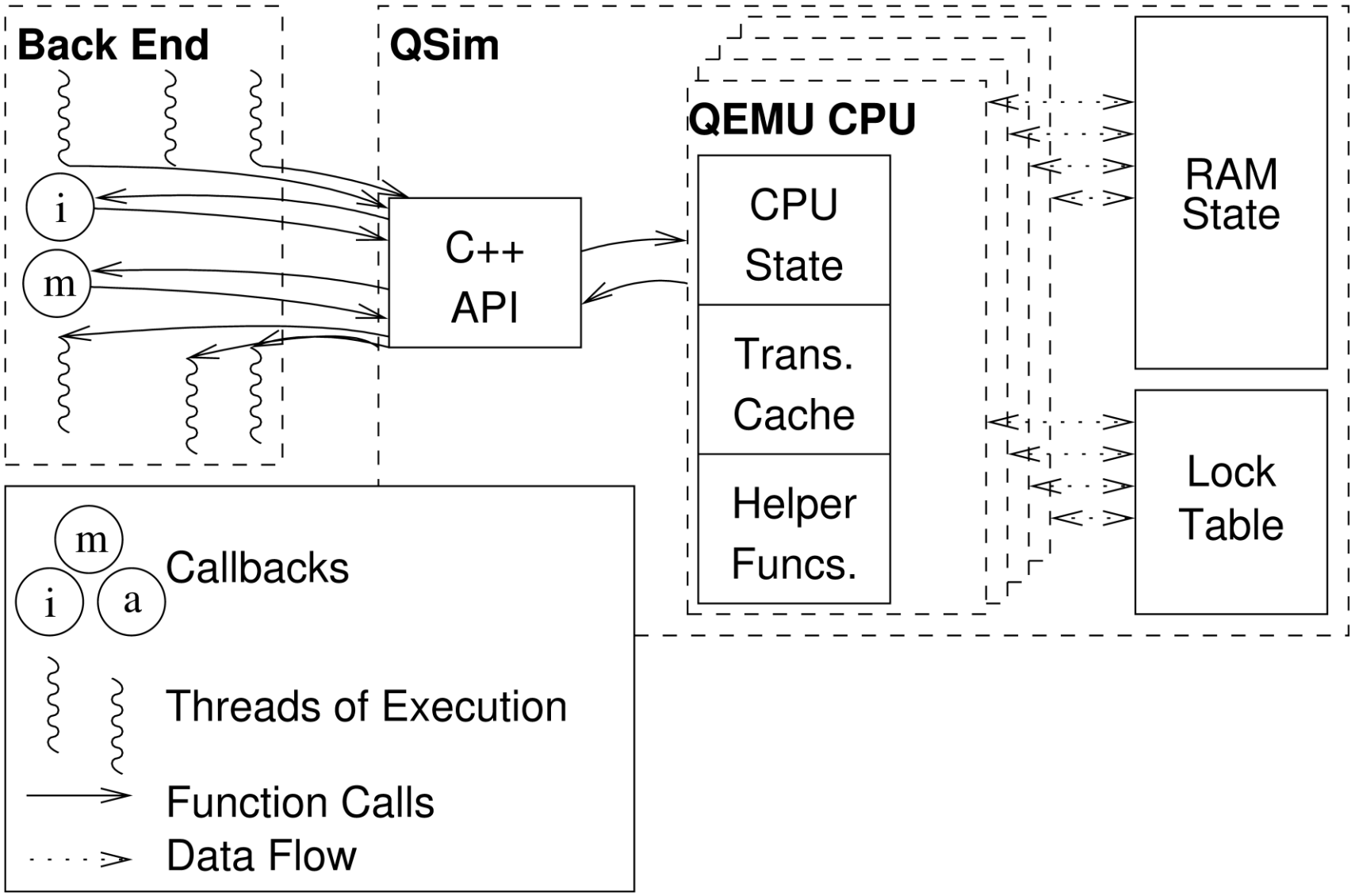


- Instrumentation of the translation cache
- Timing model feedback – backpressure from hardware events
- Synchronized advance of functional and timing models rather than roll-back and recovery
- Memory and instruction information

QSim Multicore Emulator

- Functional front-end for microarchitecture simulation
  - Runs unmodified x86 (32-bit) binaries on lightly-modified Linux kernel.
  - Provides callback interface for execution events
  - Callbacks generated for all instructions, including OS
  - Optional callbacks for instruction read during translation
    - Support for distributed memory.
- Based on QEMU dynamic binary translator
- Inserts calls to callback functions into translation cache
- Adds support for precise instruction counting
  - User-mode threading

How it Works: Block Diagram



Getting, Building, and Installing QSim

- Latest version supported by Zesto core model and remote API:

  http://www.cdkersey.com/qsim-web/releases/qsim-0.1.5.tar.bz2

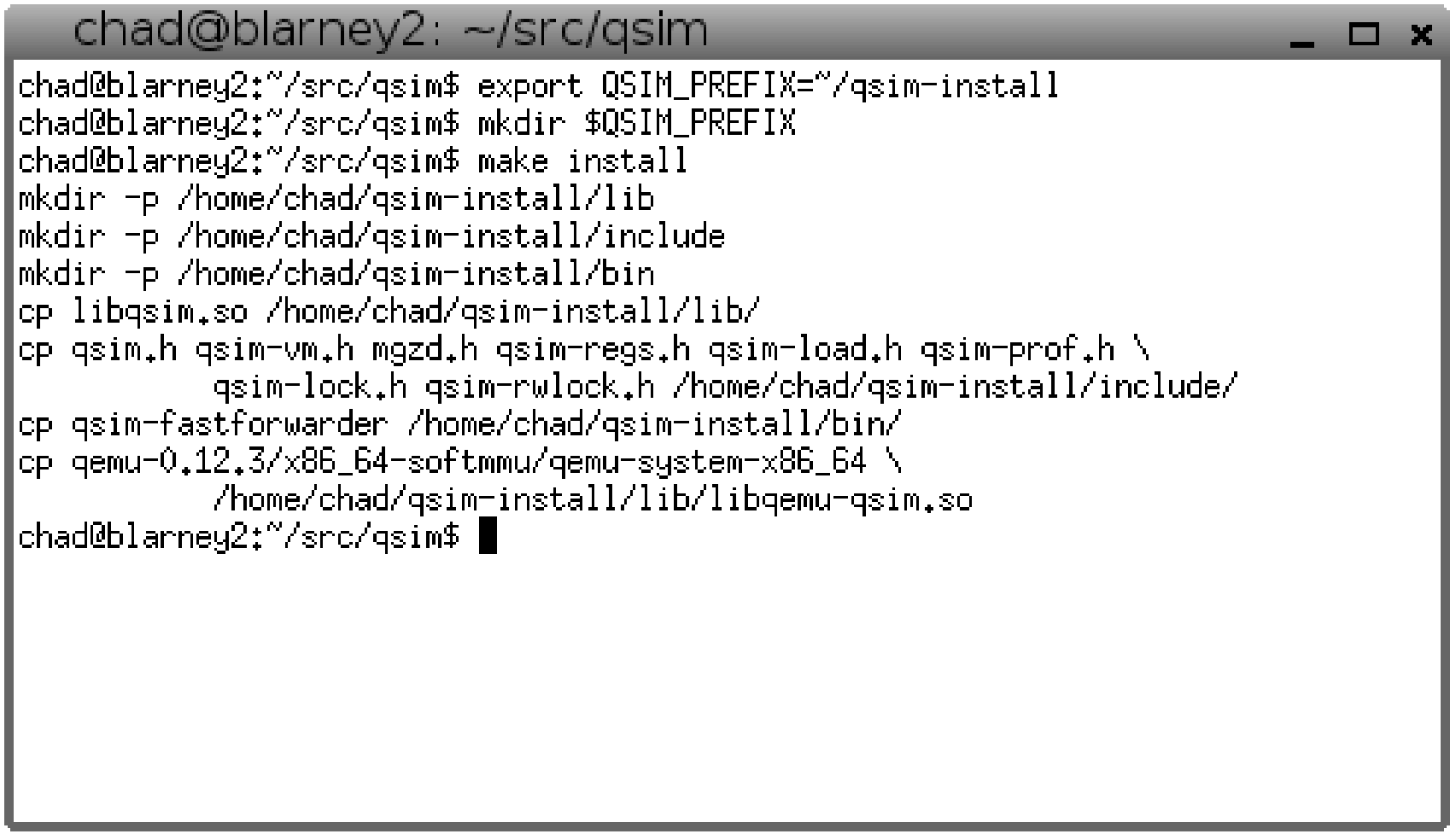
- Latest release (preferred for new core models):

  https://github.com/cdkersey/qsim/archive/0.2.1.tar.gz

- Current development tree (API is stable):

  https://github.com/cdkersey/qsim

- INSTALL file in source tree: step-by-step installation instructions.
- Benchmarks (pre-compiled applications) at:
  http://www.cdkersey.com/qsim-web/releases/qsim-benchmarks-0.1.1.tar.bz2


chad@blarney2: ~/src/qsim
chad@blarney2:~/src/qsim$ export QSIM_PREFIX=~/qsim-install
chad@blarney2:~/src/qsim$ mkdir $QSIM_PREFIX
chad@blarney2:~/src/qsim$ make install
mkdir -p /home/chad/qsim-install/lib
mkdir -p /home/chad/qsim-install/include
mkdir -p /home/chad/qsim-install/bin
cp libqsim.so /home/chad/qsim-install/lib/
cp qsim.h qsim-vm.h mgzd.h qsim-regs.h qsim-load.h qsim-prof.h \
qsim-lock.h qsim-rwlock.h /home/chad/qsim-install/include/
cp qsim-fastforwarder /home/chad/qsim-install/bin/
cp qemu-0.12.3/x86_64-softmmu/qemu-system-x86_64 \
/home/chad/qsim-install/lib/libqemu-qsim.so
chad@blarney2:~/src/qsim$

Using QSim: Application Requirements

- Applications must be:
  - Linux ELF Binaries
  - All libraries included
  - All required input data included
  - Most included benchmarks statically linked
- qsim-load expects a .tar file with a runme.sh
  - `$NCPUS` is the number of emulated HW threads

fmm/runme.sh

```
#!/sbin/ash
echo $NCPUS > ncpus
cat input.top \
  ncpus input.bot \
  > input
./FMM < input
```

barnes/runme.sh

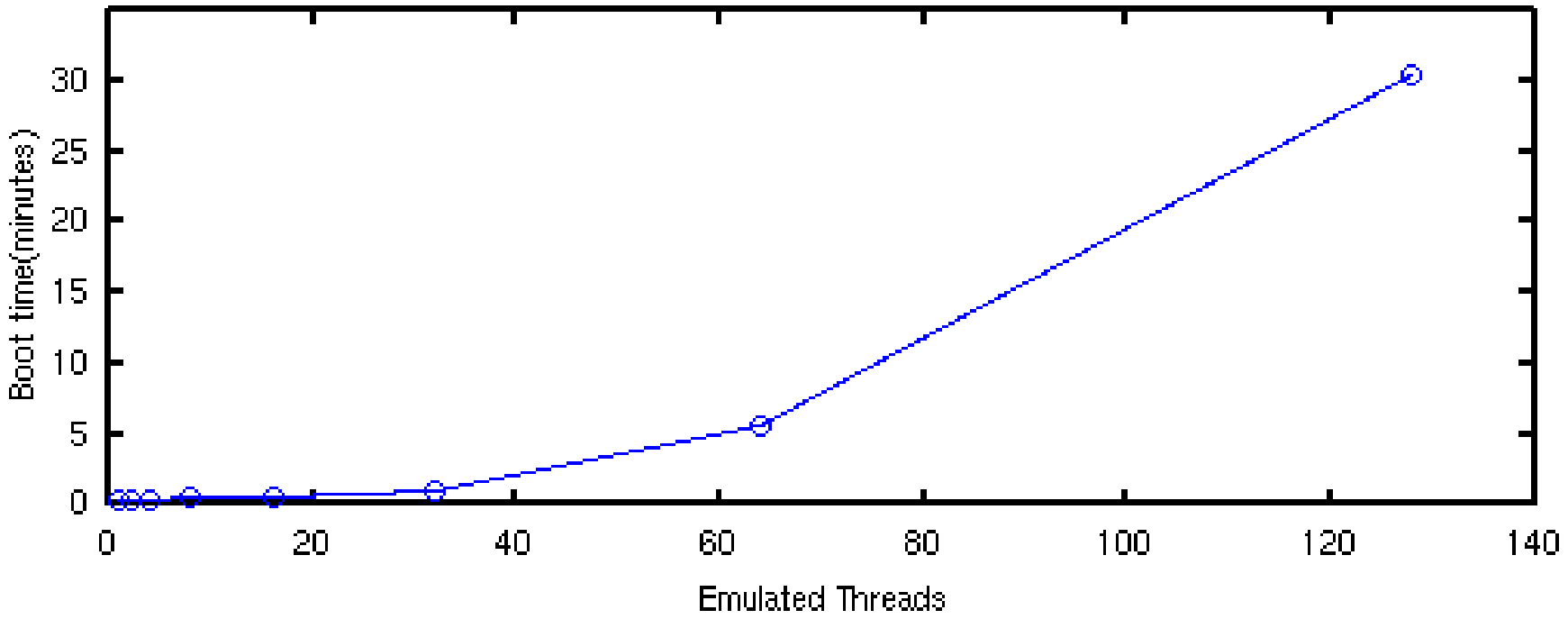
```
#!/sbin/ash
echo $NCPUS >> input
./BARNES < input
```

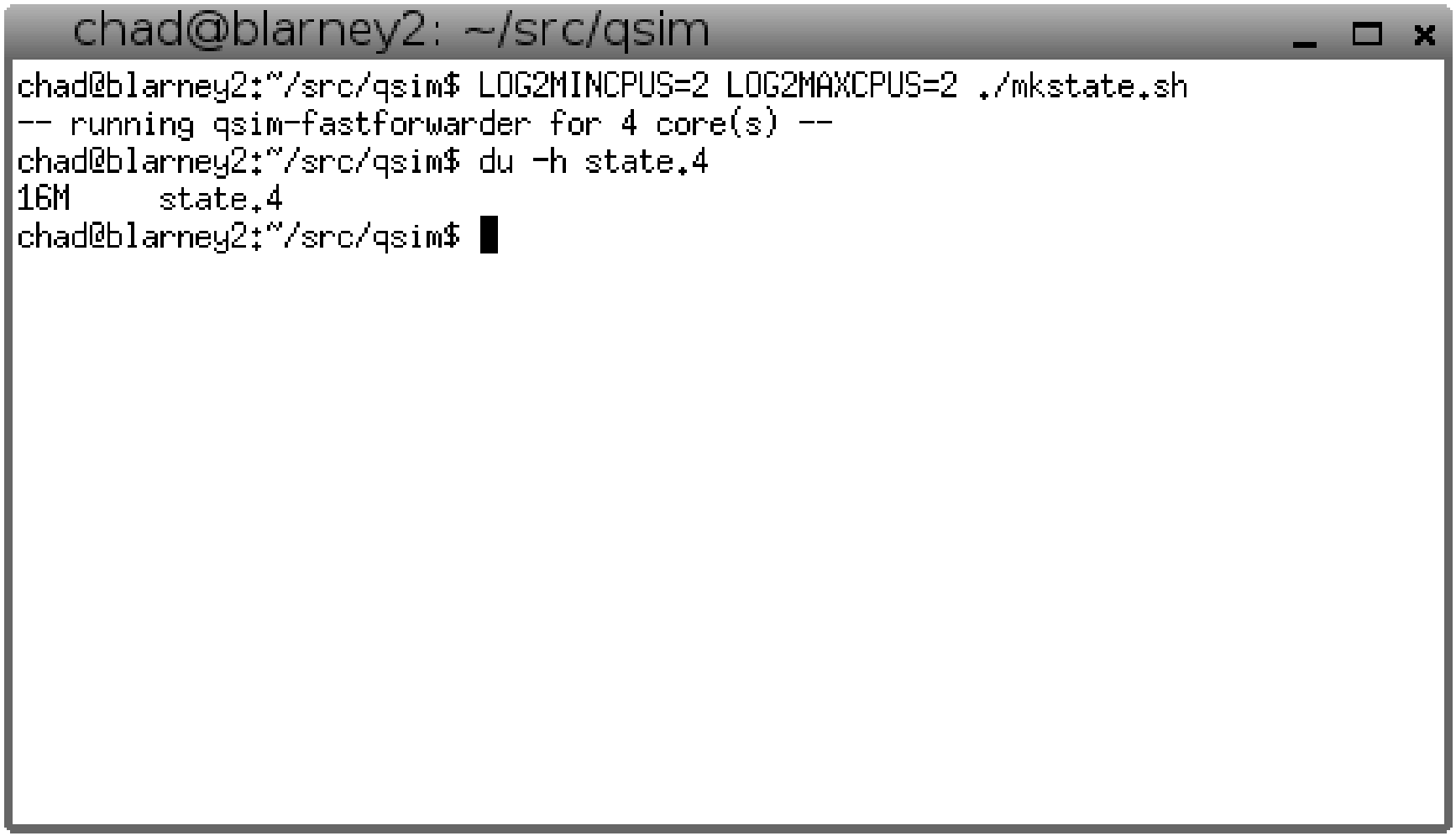
radiosity/runme.sh

```
#!/sbin/ash
./RADIOSITY \
  -ae 5000.0 \
  -en 0.050 \
  -bf 0.10 \
  -batch -room \
  -p $NCPUS
```

Using QSim: Saved States

- OS Boot for large number of CPUs takes a long time even at high (~10MIPS) simulation speeds
- Use state files to checkpoint already-booted CPU state
- mkstate.sh script for generating state files.
  - Script that runs qsim-fastforwarder program




chad@blarney2: ~/src/qsim
chad@blarney2:~/src/qsim$ LOG2MINCPUS=2 LOG2MAXCPUS=2 ./mkstate.sh
-- running qsim-fastforwarder for 4 core(s) --
chad@blarney2:~/src/qsim$ du -h state.4
16M	state.4
chad@blarney2:~/src/qsim$

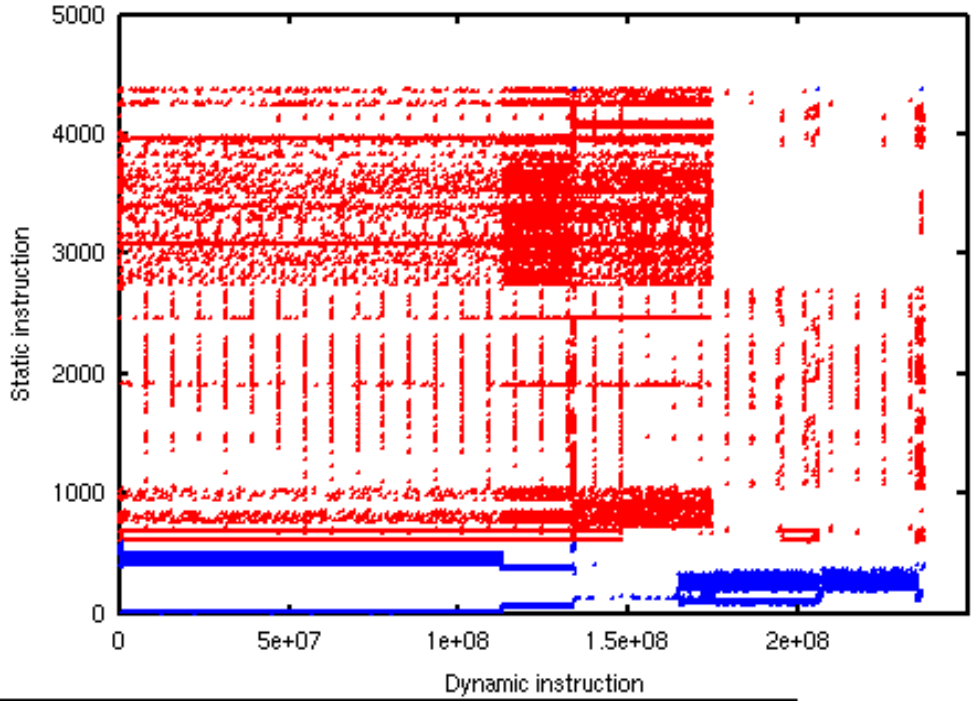
Using QSim: Application Start/End Markers

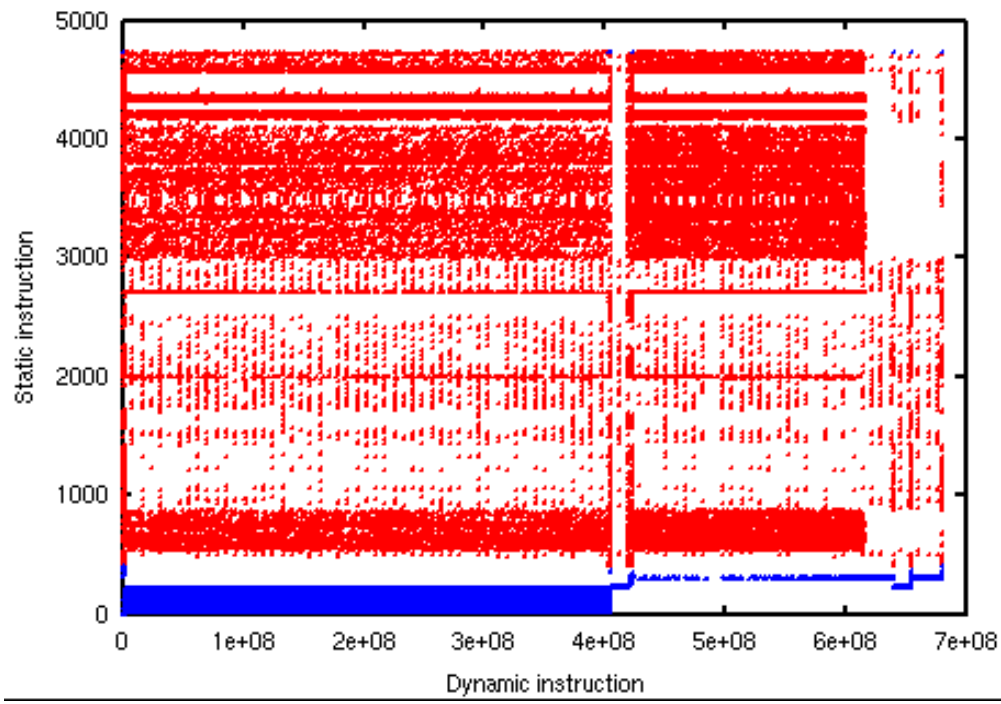
- Need to mark "region of interest" in applications.
- This is done through "magic" instructions.
- CPUID instruction with special register values:
  - `%rax = 0xaaaaaaaaa for application start`
  - `%rax = 0xfa11dead for application end`

```
#ifdef QSIM
#define APP_START() do { \
   __asm__ __volatile__("cpuid;"::"a"(0xaaaaaaaaa));\
   } while(0)

#define APP_END() do { \
   __asm__ __volatile__("cpuid;"::"a"(0xfa11dead));\
   } while(0)
#endif
```

Using QSim: Applications and the OS

- "Emulates all OS instructions" means exactly that:
  - Timer interrupts are necessary and the scheduler does run.
  - Quite a bit lot of time is spent handling page faults.
- Can query and filter out OS instructions if they are considered irrelevant:
  - `OSDomain::getProt(core) == OSDomain::PROT_KERNEL`

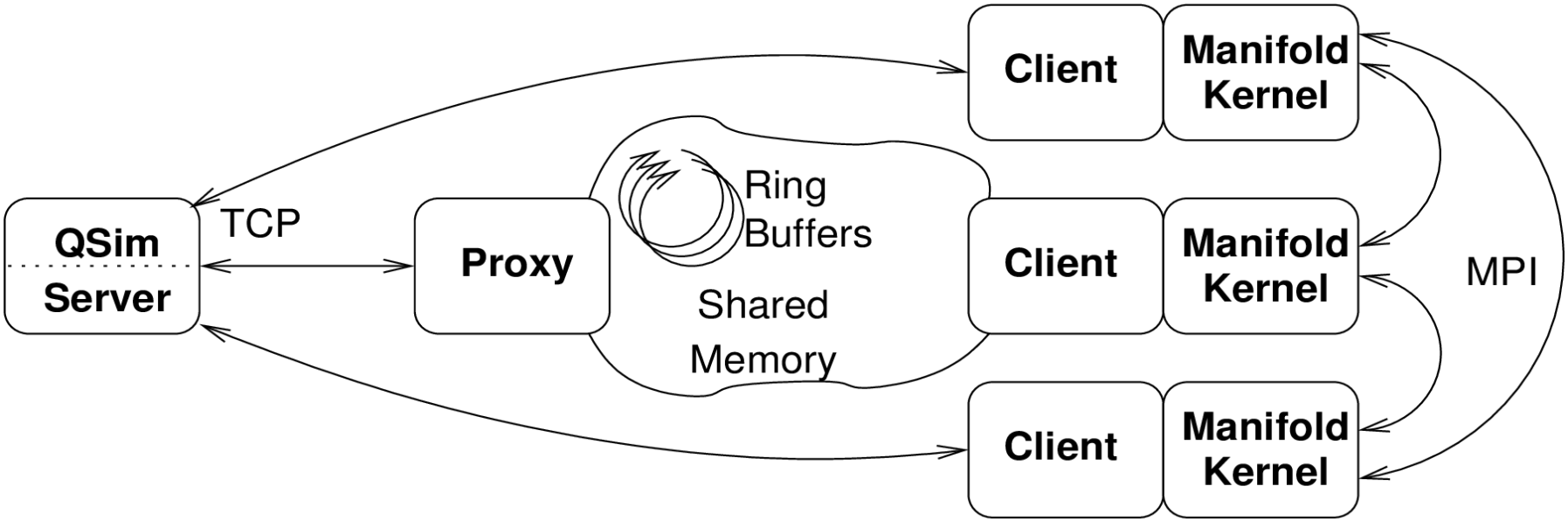


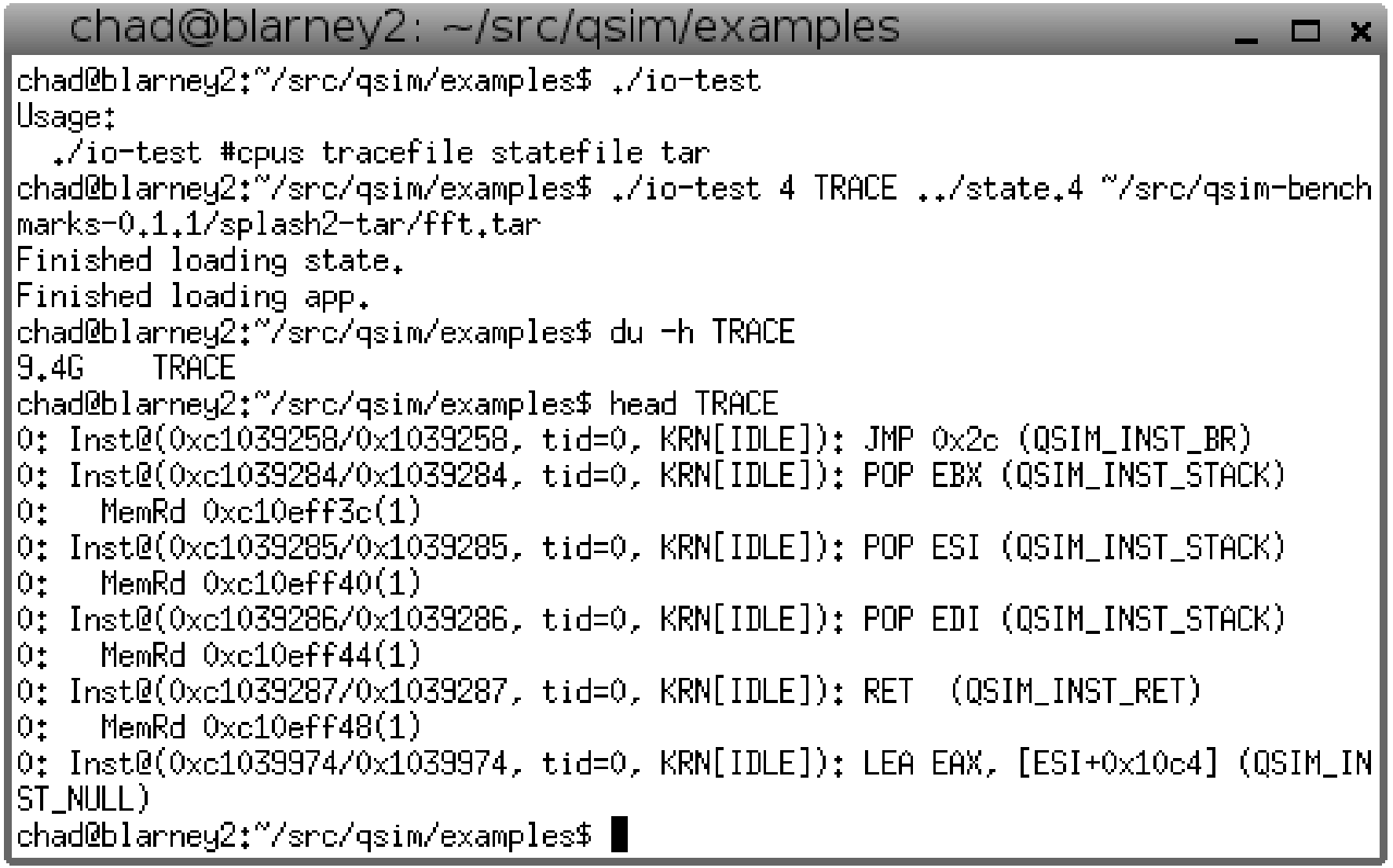


OS USER

QSim Remote API: The Server and Proxy

- A subset of the QSim API is provided for operation over a network.
- Server performs functional simulation; client performs timing model.
- Client API looks like QSim API:
- One thread per client.
- Proxy acts as intermediary between clients and server.
  - Third program that hides network latency.
  - Necessary because client library uses blocking I/O.




chad@blarney2: ~/src/qsim/examples
chad@blarney2:~/src/qsim/examples$ ./io-test
Usage:
./io-test #cpus tracefile statefile tar
chad@blarney2:~/src/qsim/examples$ ./io-test 4 TRACE ../state.4 ~/src/qsim-benchmarks-0.1.1/splash2-tar/fft.tar
Finished loading state.
Finished loading app.
chad@blarney2:~/src/qsim/examples$ du -h TRACE
9.4G TRACE
chad@blarney2:~/src/qsim/examples$ head TRACE
0: Inst@(0xc1039258/0x1039258, tid=0, KRN[IDLE]): JMP 0x2c (QSIM_INST_BR)
0: Inst@(0xc1039284/0x1039284, tid=0, KRN[IDLE]): POP EBX (QSIM_INST_STACK)
0: MemRd 0xc10eff3c(1)
0: Inst@(0xc1039285/0x1039285, tid=0, KRN[IDLE]): POP ESI (QSIM_INST_STACK)
0: MemRd 0xc10eff40(1)
0: Inst@(0xc1039286/0x1039286, tid=0, KRN[IDLE]): POP EDI (QSIM_INST_STACK)
0: MemRd 0xc10eff44(1)
0: Inst@(0xc1039287/0x1039287, tid=0, KRN[IDLE]): RET (QSIM_INST_RET)
0: MemRd 0xc10eff48(1)
0: Inst@(0xc1039974/0x1039974, tid=0, KRN[IDLE]): LEA EAX, [ESI+0x10c4] (QSIM_INST_NULL)
chad@blarney2:~/src/qsim/examples$

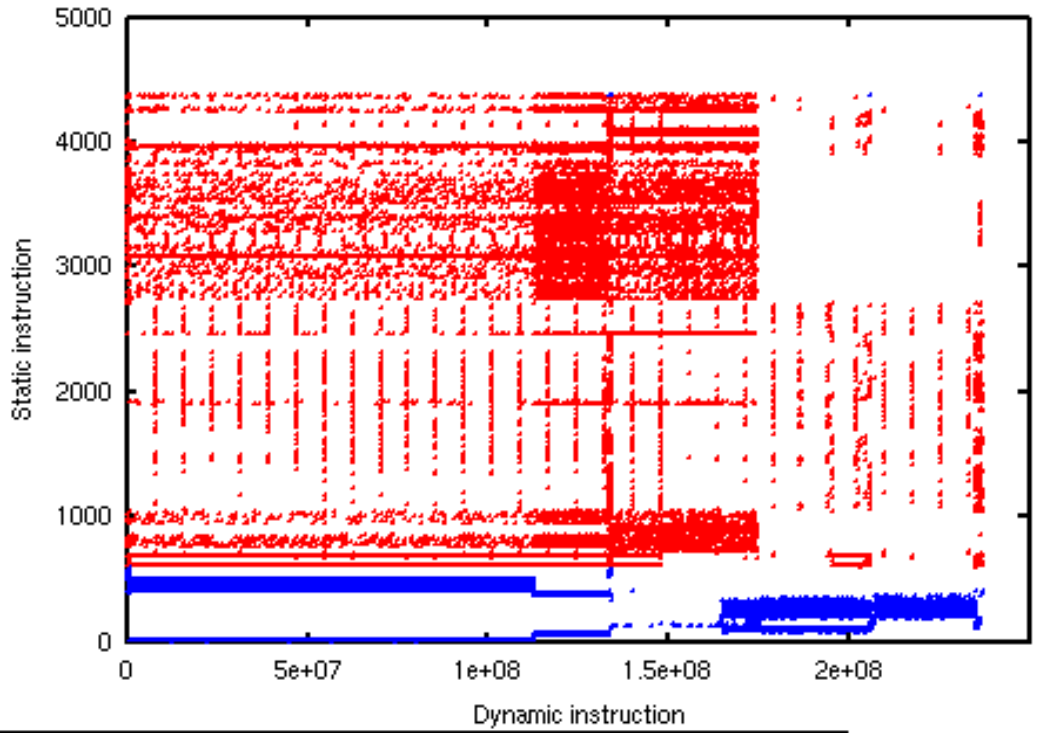
QSim API: Basic Functionality

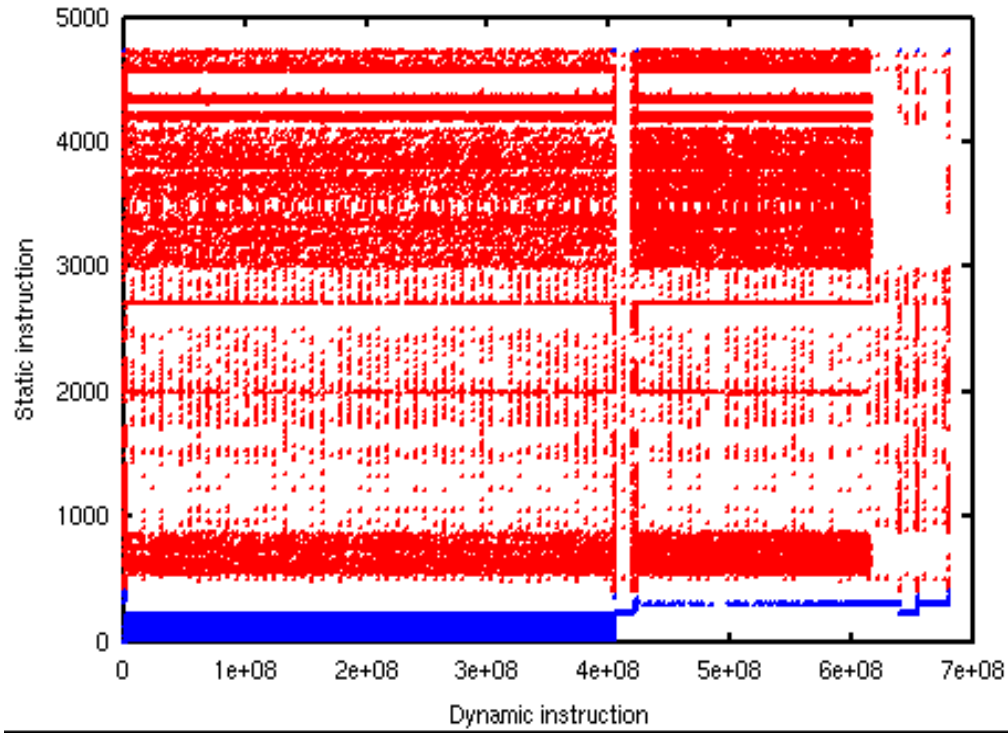
OSDomain Func.	Purpose
get_prot(core)	Get protection level (KERN, USER)
get_idle(core)	Is the core running the kernel idle loop?
run(core, *n*)	Run the selected core for *n* instructions.
timer_interrupt()	Interrupt all CPUs. (required)
Callback Setters	
set_mem_callback(obj, *f*)	Memory accesses (loads, stores)
set_atomic_callback(obj, *f*)	Atomic read/modify/write operations
set_inst_callback(obj, *f*)	All instructions
set_reg_callback(obj, *f*)	Register reads/writes
set_int_callback(obj, *f*)	Interrupts, including exceptions
set_trans_callback(obj, *f*)	Reads into QEmu translation cache
set_app_start_callback(obj, *f*)	Beginning of application (start marker)
set_app_end_callback(obj, *f*)	End of application (end marker)

For full details, including callback prototypes, see user guide.

QSim API: Timer Interrupt

- Information about time not communicated back to QSim from the timing model.
  - Periodic interrupts simulating PIT events are the only timing information that reaches the OS.
  - `OSDomain::timer_interrupt()` must be called periodically (~1-10ms to sim time) for the entire `OSDomain`
- Not setting the timer interrupt leads to scheduled threads never running.





OS USER

QSim API: Simple Sample Program

```
class TraceWriter
  {private: OSDomain &osd;  ostream &trf;
   public:  bool done;
    TraceWriter(OSDomain &osd, ostream &trf) :
      osd(osd), trf(trf), done(false)
    { osd.set_inst_cb(this, &TraceWriter::inst_cb);
      osd.set_app_end_cb(this, &TraceWriter::app_end_cb); }
    void inst_cb(int c, uint64_t v, uint64_t p, uint8_t l,
                 const uint8_t *b, enum inst_type t)
    { trf << std::dec << c << ',' << v << ',' << p << ',' << t << endl; }
    int app_end_cb(int c) { done = true; return 1; } };

  int main(int argc, char** argv)
  { ofstream trace(argv[3]); // Arg 3 : trace file
    OSDomain osd(argv[1]);   // Arg 1 : state file
    load_file(osd, argv[2]); // Arg 2 : benchmark
    TraceWriter tw(osd, trace);
    while (!tw.done)
    { for (unsigned i = 0; i < 100; ++i)
        for (unsigned j = 0; j < osd.get_n(); ++j)
          osd.run(j, 10000);
      osd.timer_interrupt(); }
    trace.close();
    return 0; }
```

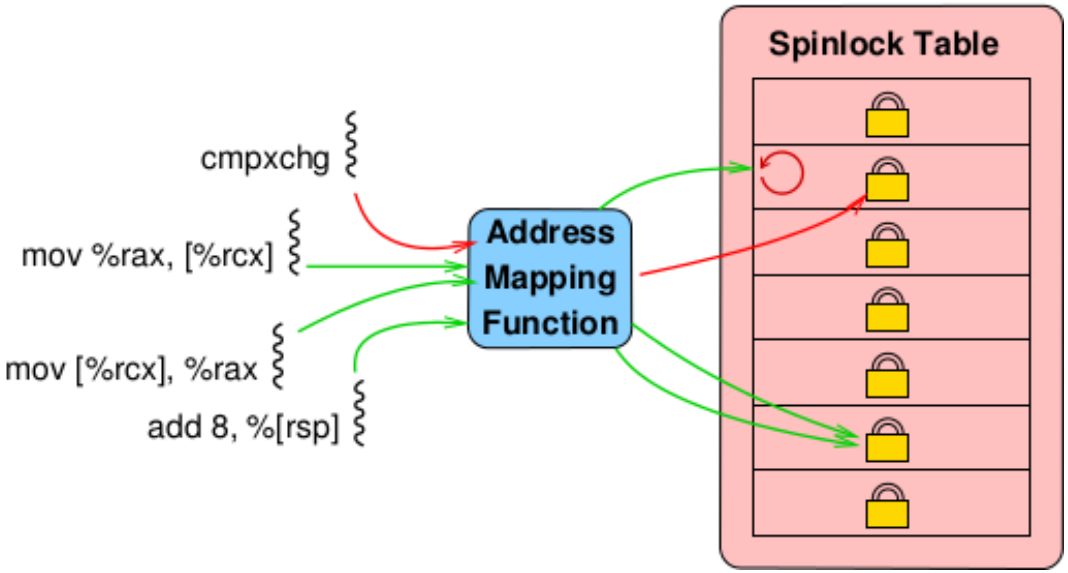
Outline

- Introduction
- Execution Model and System Architecture
- Multicore Emulator Front-End
- Component Models
  - Cores
  - Network
  - Memory System
- Building and Running Manifold Simulations
- Physical Modeling: Energy Introspector
- Some Example Simulators

Additional Slides

How it Works: Lock Table

- QSim is a thread-safe library.
- Each simulated CPU is a QEmu instance.
- RAM state is shared.
- Memory has readers/writer lock semantics
  - 1 thread executing an atomic OR
  - any number of threads executing ordinary memory operations
- This can be applied on a per-address basis
- We build a table of spinlocks
  - Spinlock code appropriated from Linux kernel.



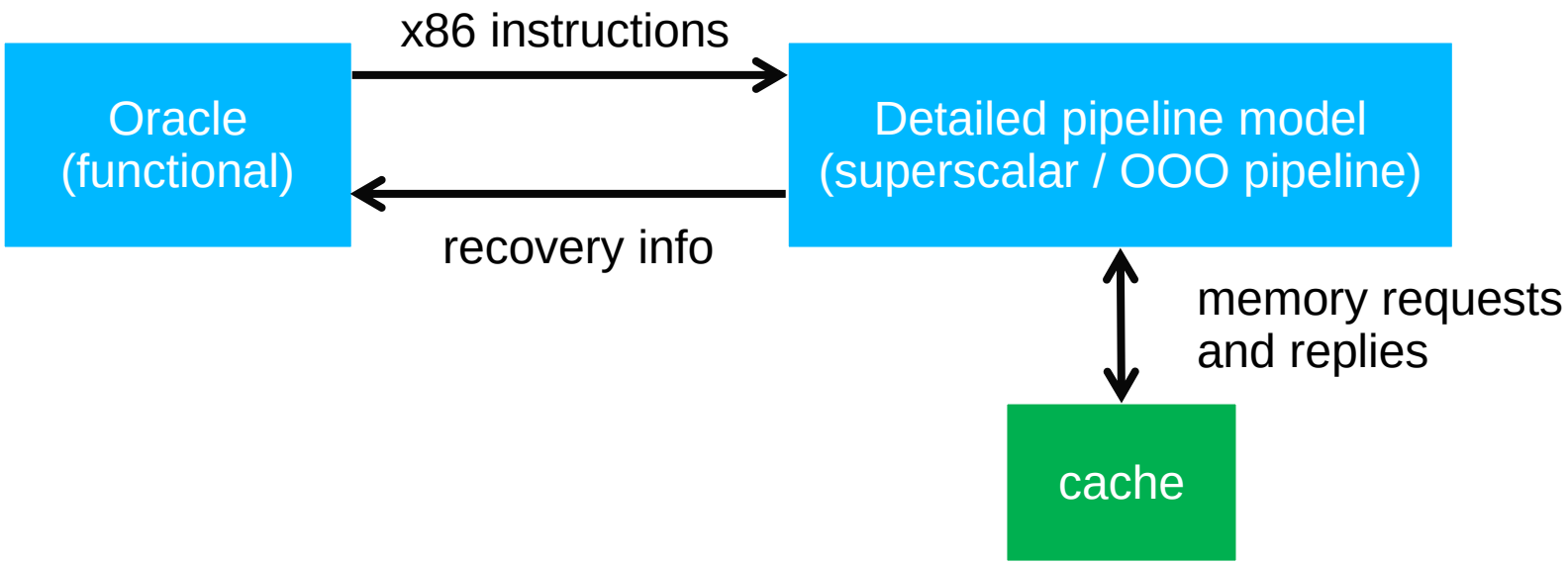
Back-end Timing Models

- Core Models
- Interconnection Network
- Memory System
  - Coherence Cache Hierarchy
  - DRAM Controller

Core Models

- Zesto – cycle-level x86 processor model
- SPX – a light pipe-lined model
- SimpleProc – a 1-IPC model

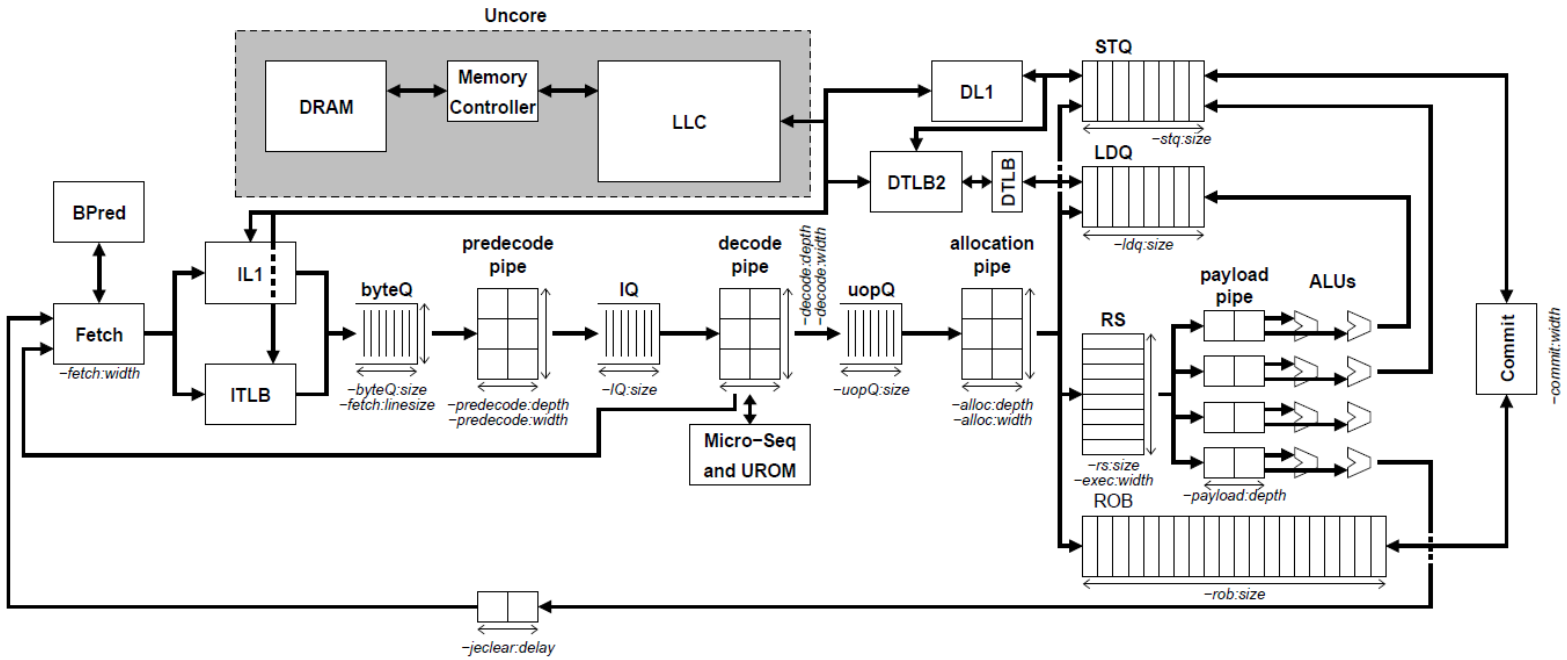
Zesto* Overview



- Consists of an oracle and a detailed pipeline
- Oracle is an "execution-at-fetch" functional simulator; fetched instructions are first passed to oracle
- Pipeline has 5 stages
- Very detailed; slow (10's of KIPS)
- Ported to Manifold as a component

*G. Loh, etc., "Zesto: a cycle-level simulator for highly detailed microarchitecture exploration," *International Symposium on Performance Analysis of Software and Systems (ISPASS)*, pp53-64, 2009

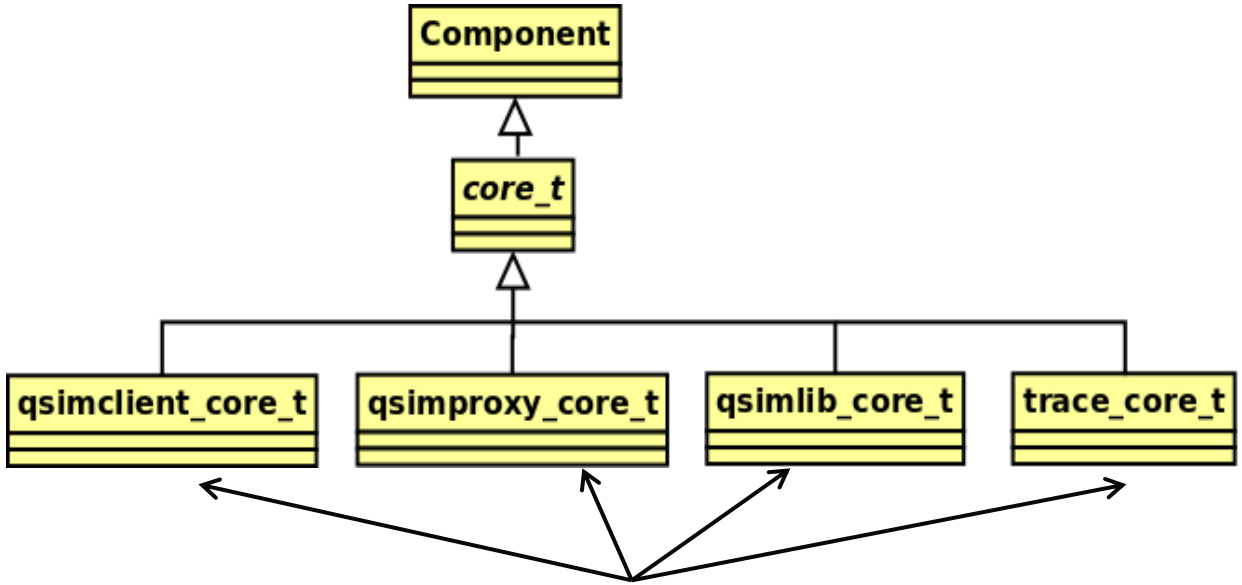
Zesto



- X86 based timing model derived from Zesto cycle-level simulator
  - IA 32 support

Zesto Interface

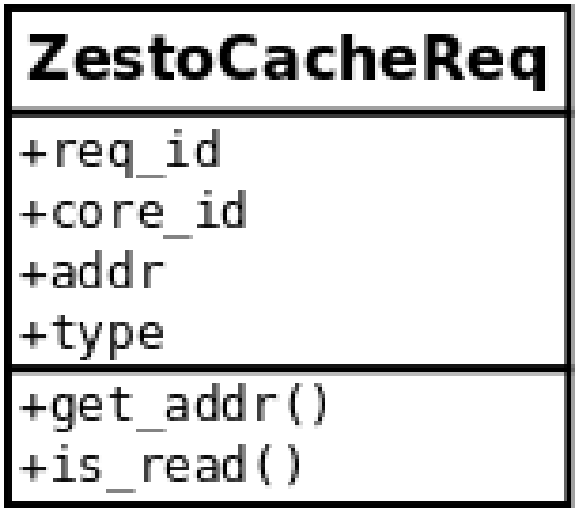
- Front-end interfaces
- Cache request
- Event handler (for cache response)
- Clocked function



Multiple instances with different front-end interfaces

Zesto Interface

- ZestoCacheReq



- event handler for cache response

```
class core_t {
public:
  ...
  void cache_response_handler(int, ZestoCacheReq*);
  ...
};
```

Zesto Interface

- Clocked function
  - must be registered with a clock
  - Zesto does not register this function, so it must be registered in the simulator program.

```
void core_t :: tick()
{
    //get the pipeline going
    commit->step();
    exec->LDST_exec();
    ...
    alloc->step();
    decode->step();
    fetch->step();
    ...
}
```

Simplified Pipeline eXecution (SPX)

<*A simpler, lighter core model for out-of-order and in-order pipelines*>

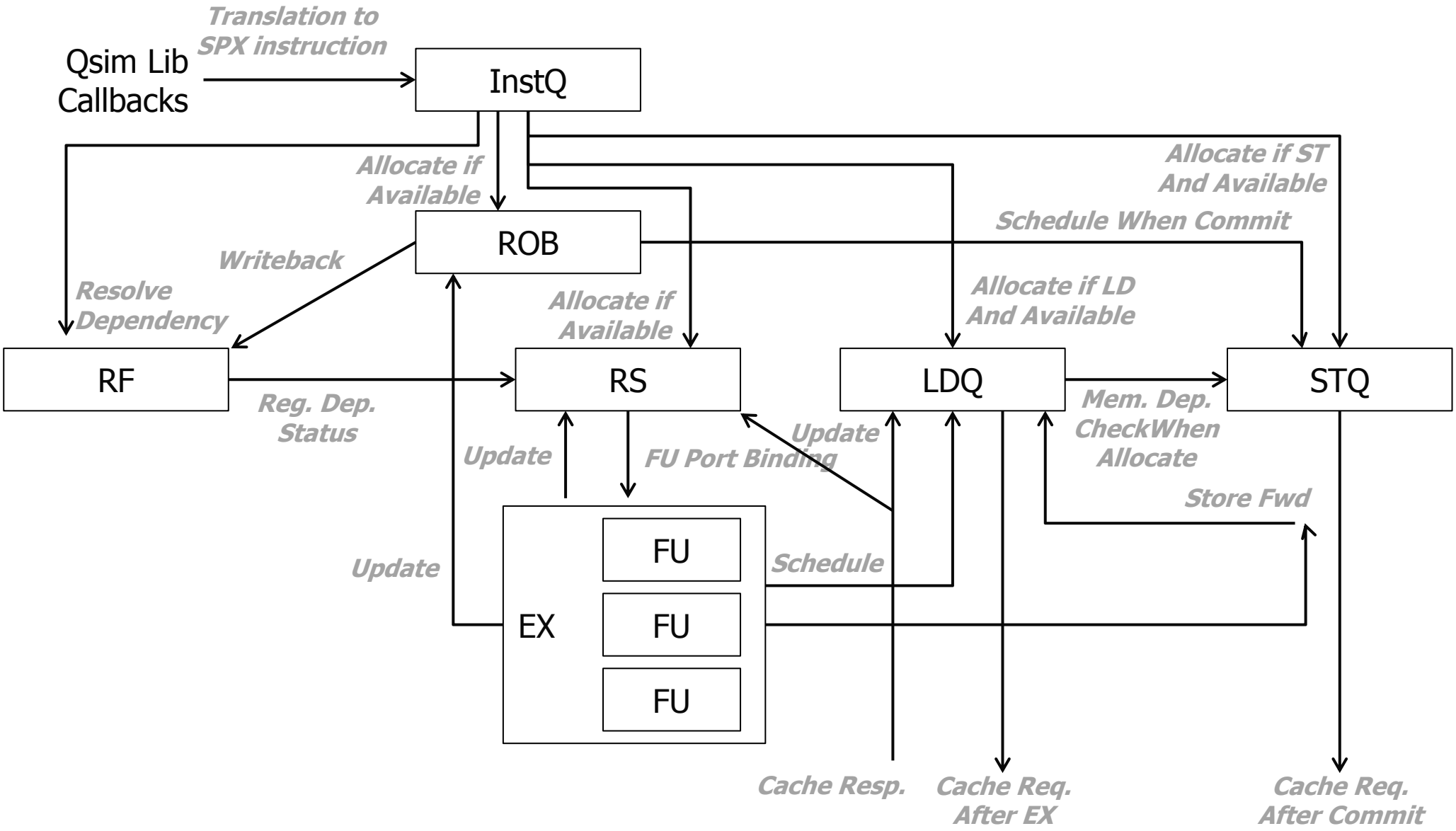
SPX

- SPX is an *abbreviated core model* that support both *out-of-order* and *in-order* executions.
- Purpose: *Detail* vs. *Speed*
  - For *less core-sensitive simulations* (i.e., memory/network traffic analysis), detailed core implementation may be unnecessary.
  - *Physics simulations* need longer observation time (in real seconds) than typical architectural simulations
    - e.g., temperature doesn't change much for several hundreds million clock cycles (or several hundred milliseconds) of simulation.
- The SPX model provides enough details to model *out-of-order* and *in-order* executions, and other *optional behaviors* are all abbreviated (e.g., predictions)
- The SPX requires *Qsim Library* for detailed instruction information.
  - *Queue model* is not currently supported.
  - SPX uses direct *Qsim callback* functions.

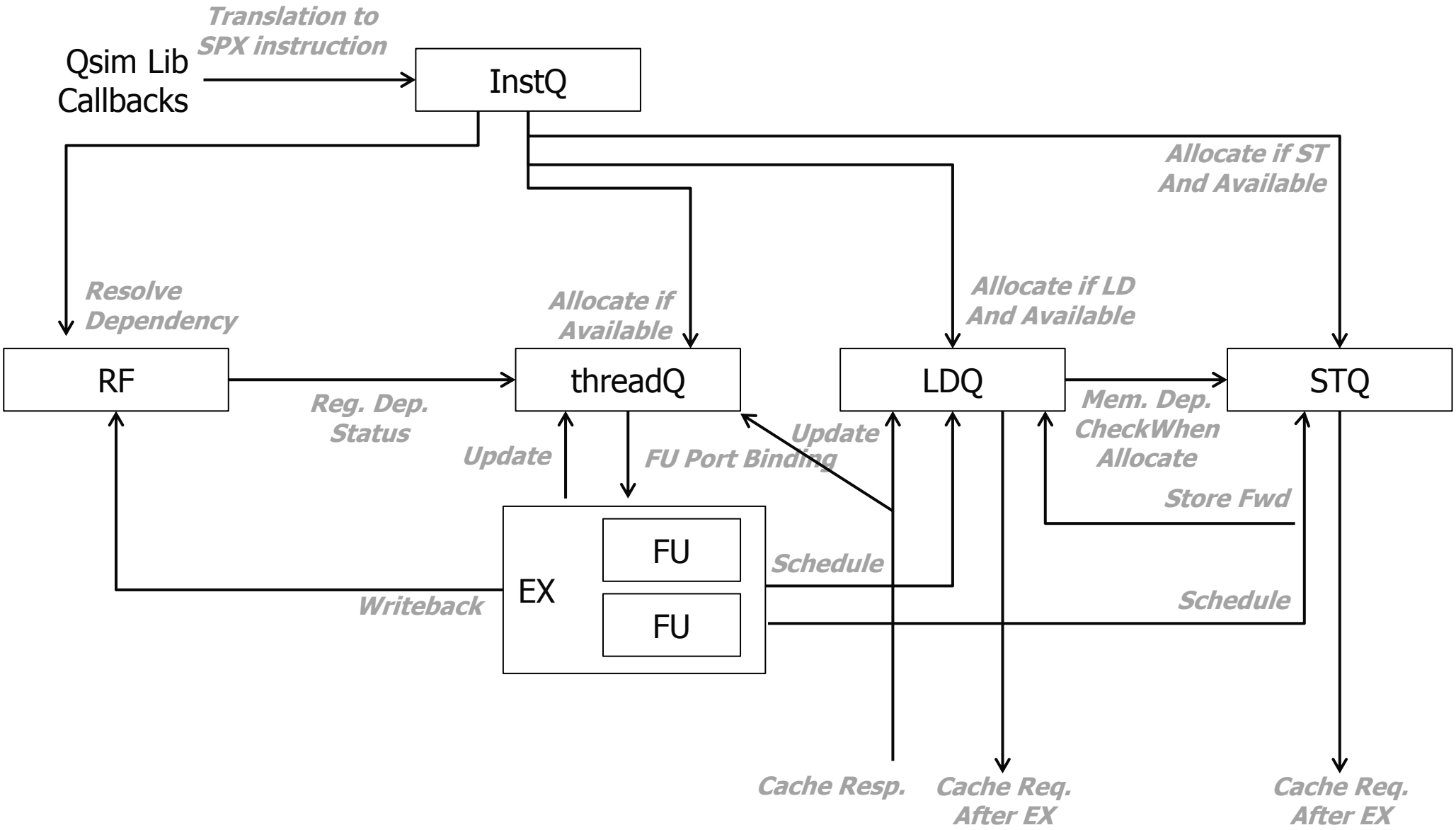
Modeled Components in SPX

- **InstQ**: A queue that fetches the instructions from Qsim.
- **RF**: Register file that tracks only dependency; behaves more like *RAT*.
  - Dependency for *general register files* and *flags* are maintained.
- **RS**: Dependency status tracker.
  - It does not have an actual table like scoreboard.
  - When an instruction is ready (cleared from dependency), it is put in the *ReadyQ*.
  - In-order execution is modeled with 1-entry RS.
- **EX (FU)**: Multi-ported execution units.
  - Each FU is defined with *latency* and *issue rate*.
- **ROB**: Re-order buffer for out-of-order execution
  - An instruction can be broken into *multiple sub-instructions* (e.g., u-ops), and ROB commits the instruction when all sub-instructions are completed.
- **LDQ/STQ**: Load and store queues
  - These queues check *memory dependency* between ST/LD instructions.

Out-of-order Pipeline

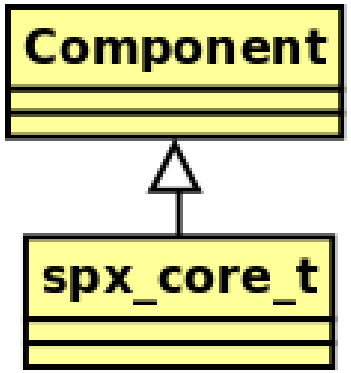


In-order Pipeline



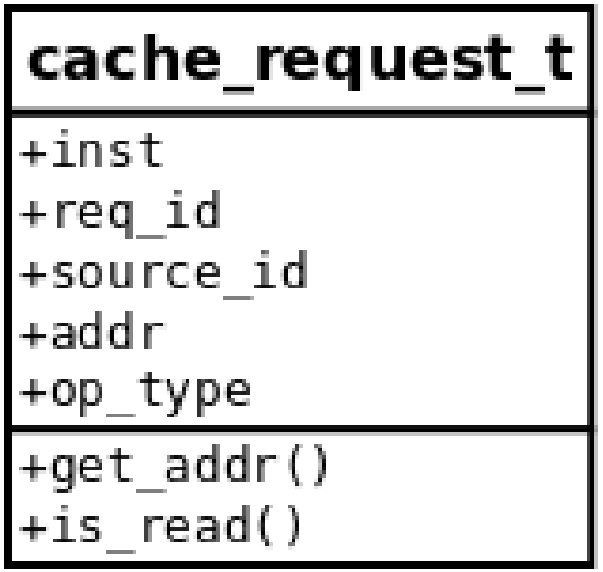
SPX Interface

- Front-end
- Cache request
- Event handler (for cache response)
- Clocked function



SPX Interface

- cache_reqest_t



- event handler for cache response

```
class spx_core_t {
public:
  ...
  void handle_cache_response(int, cache_reqest_t*);
  ...
};
```

- clocked function
  - must be registered with a clock
  - SPX does not register this function, so it must be registered in the simulator program.

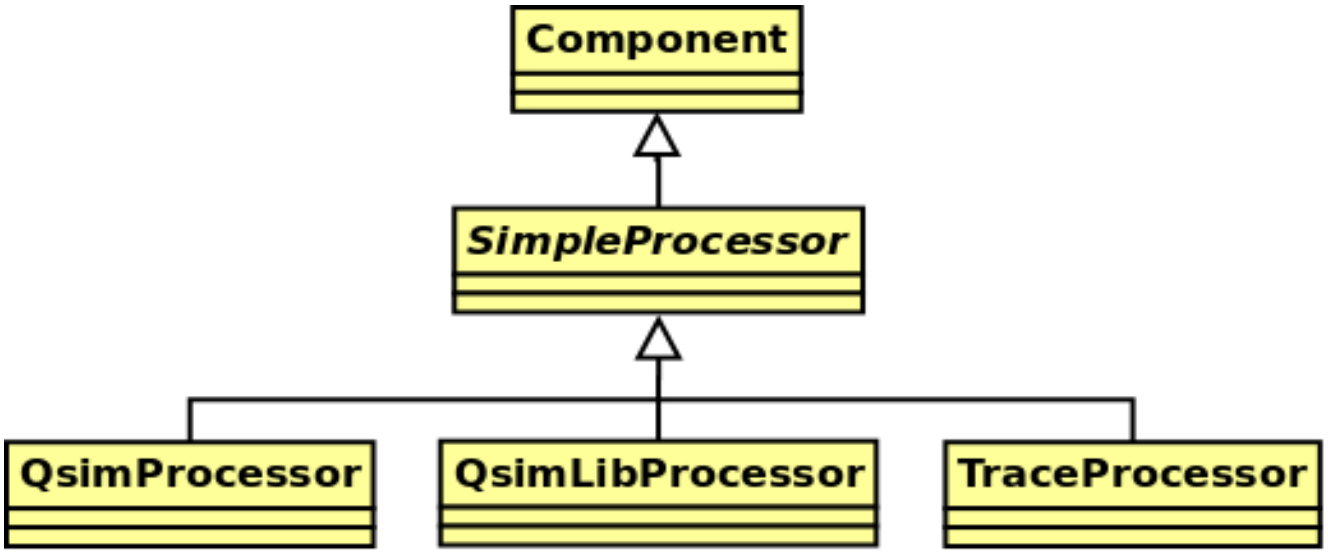
```
void spx_core_t :: tick()
{
    ...
}
```

SimpleProc

- 1-IPC: in general one instruction is executed every cycle
- MSHR limits number of outstanding cache requests

SimpleProc Interface

- Front-end
- Cache request
- Event handler (for cache response)
- Clocked function



SimpleProc Interface

- CacheReq

- event handler for cache response

```
class SimpleProcessor {
public:
  ...
  void handle_cache_response(int, CacheReq*);
  ...
};
```

- clocked function
  - must be registered with a clock
  - SimpleProc does not register this function, so it must be registered in the simulator program.

```
void SimpleProcessor :: tick()
{
    ...
}
```

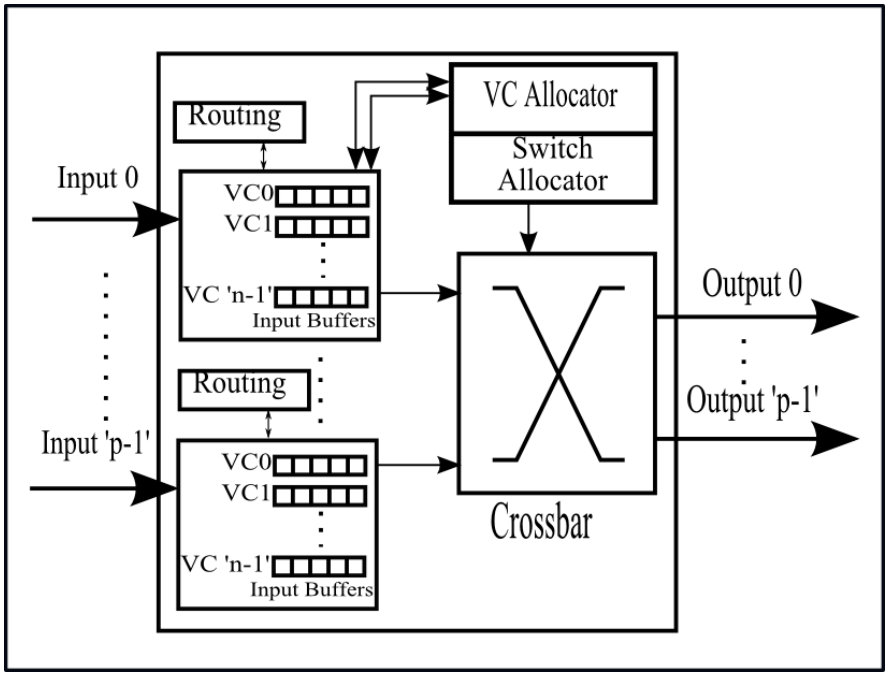
Outline

- Introduction
- Execution Model and System Architecture
- Multicore Emulator Front-End
- Component Models
  - Cores
  - Network
  - Memory System
- Building and Running Manifold Simulations
- Physical Modeling: Energy Introspector
- Some Example Simulators

Back-end Timing Models

- Core Models
- Interconnection Network
- Memory System
  - Coherence Cache
  - DRAM Controller

The Network Simulator - IRIS



- Virtual channels
- Ring and torus
- Request-reply network
- Single-flit and multiflit packets
- Credit-based flow control
- Must instantiate a network interface

- Basic router/switch designed to enable design space exploration
  - Ease of changing VC & switch allocators, buffering, routing function, selection functions, etc.
- Topology generators for Ring and Tori
  - You can write your own

Iris Interface

- User creates an Iris network using one of the following:

```
template<typename T> Ring<T>* create_ring();
template<typename T> Torus<T>* create_torus();
```

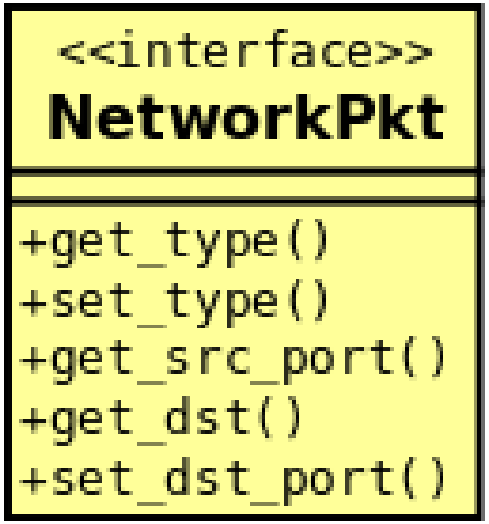
- parameters:
  - Clock& clk – clock for the network
  - ring_init_params* or torus_init_params* - parameters for the ring or torus
  - Terminal_to_net_mapping* - object that maps terminal address to network address
  - SimulatedLen<T>* - object that computes a packet's simulated length
  - VnetAssign<T>* - object that determines a packet's virtual network
  - int ni_credit_type – type of credit messages between NI and terminal
  - vector<int>* node_lp – LP assignment of the routers

Iris Interface

- NIs and routers are created internally
- NIs and routers are registered to the clock internally
- NIs can be accessed through get_interface_id(), or get_interfaces()
- Each NI can connect to one terminal (e.g., cache)
- Router is encapsulated

Iris Network Interface

- A template class:
  - `template<typename T> class GenNetworkInterface`
  - T is the type of packets sent to the network
- Network packet interface



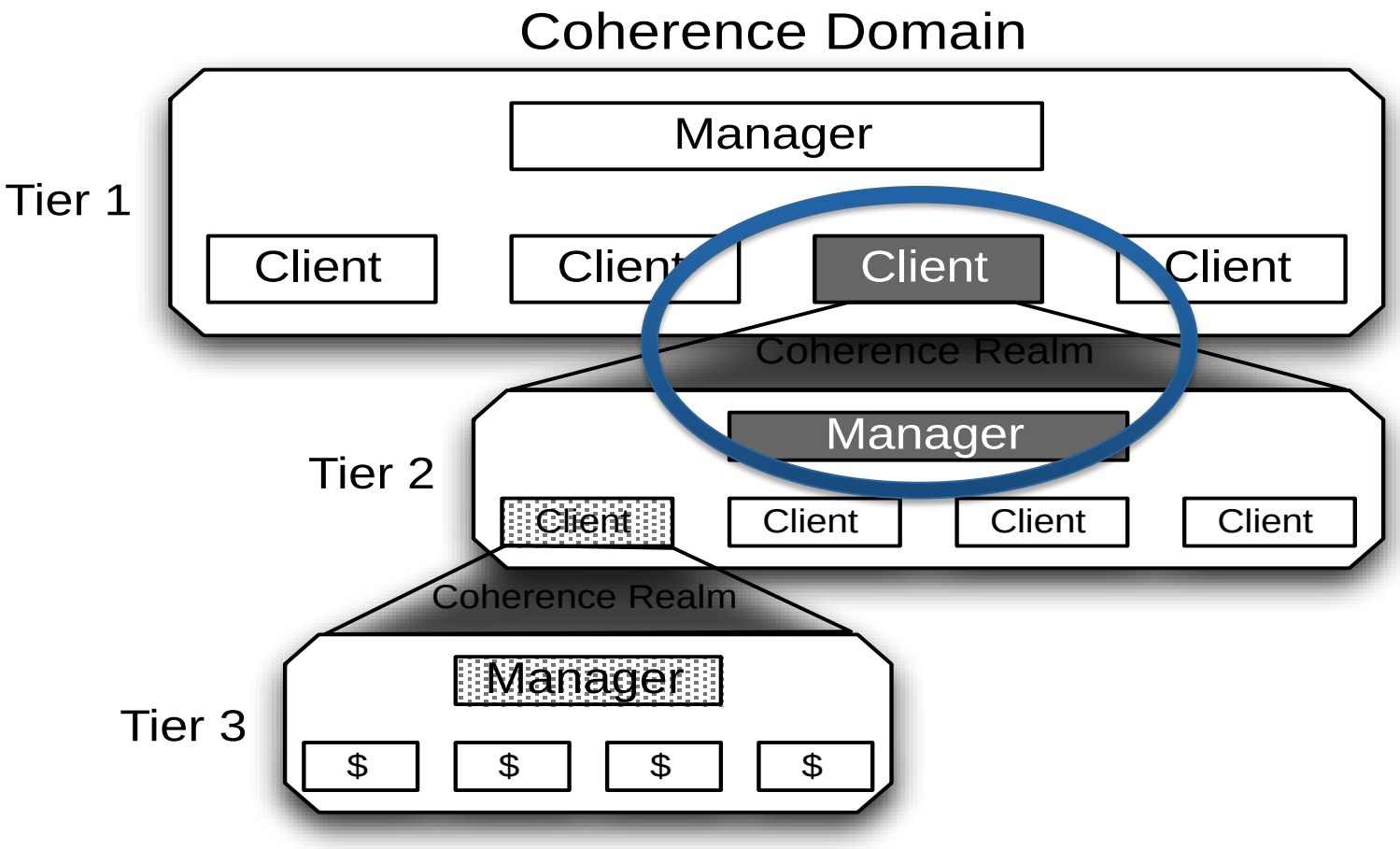
Outline

- Introduction
- Execution Model and System Architecture
- Multicore Emulator Front-End
- Component Models
  - Cores
  - Network
  - Memory System
- Building and Running Manifold Simulations
- Physical Modeling: Energy Introspector
- Some Example Simulators

Multi-core Systems and Coherence Hierarchies

- Coherence hierarchy issues
  - Several architecture specific implementations
  - Design complexity
    - Complex hierarchy state encodings
    - Many more transient states
- Solution: Coherence Realm* Encapsulation
  - Define communication interface between users and monitors
  - Enables layering of coherence
  - **Enables heterogeneity within a protocol**

Manager-Client Pairing (MCP)*



*J. G. Beu, M. C. Rosier and T. M. Conte, “Manager-Client Pairing: A Framework for Implementing Coherence Hierarchies,” Proceedings of the 44th Annual International Symposium on Microarchitecture (MICRO-44), (Porto Alegre, Brazil), Dec., 2011.

Division of Labor

- Client Agent (think cache)
  - Permission holder (Coherence State)
  - Obtains permission via acquire requests
  - **Act as a gateway in hierarchical coherence** (see algorithm)
- Manager Agent (think directory)
  - Monitor of coherence realm
    - Records sharers, owner, etc.
  - Manages permission propagation
    - Process acquire requests
    - Allocates/de-allocates permissions to/from clients
    - **Handles external requests from other realms**

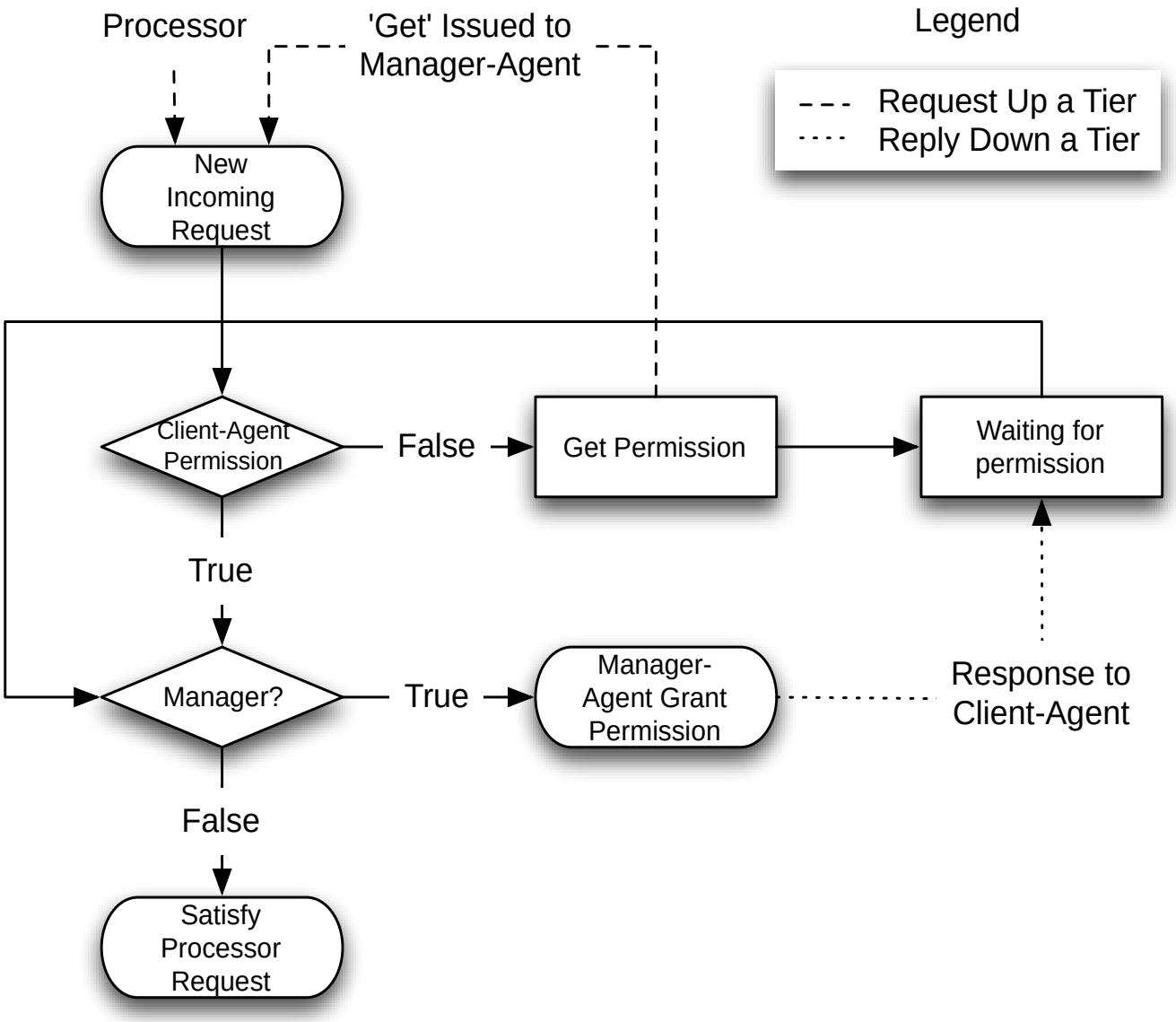
Base Functions

Agent	Responsibility	Command	Description
Client	Permission Query	ReadP	Have read permission?
		WriteP	Have write permission?
		EvictP	Have eviction permission?
	Permission Acquire	GetRead	Get read permission
		GetWrite	Get write permission
		GetEvict	Get eviction permission
	Permission Forward	FwdGrantR	Forward read permission
		FwdGrantW	Forward write permission
	Acknowledgements	AckRead	Read permission complete
		AckWrite	Write permission complete
		AckEvict	Eviction permission complete
Directory	Permission Response	GrantRead	Grant read permission
		GrantWrite	Grant write permission
		GrantEvict	Grant eviction permission
	Downgrade	DwnInval	Downgrade to invalid
		DwnRead	Downgrade to read
	Request Forwarding	FwdRead	Forward read permission
		FwdWrite	Forward write permission

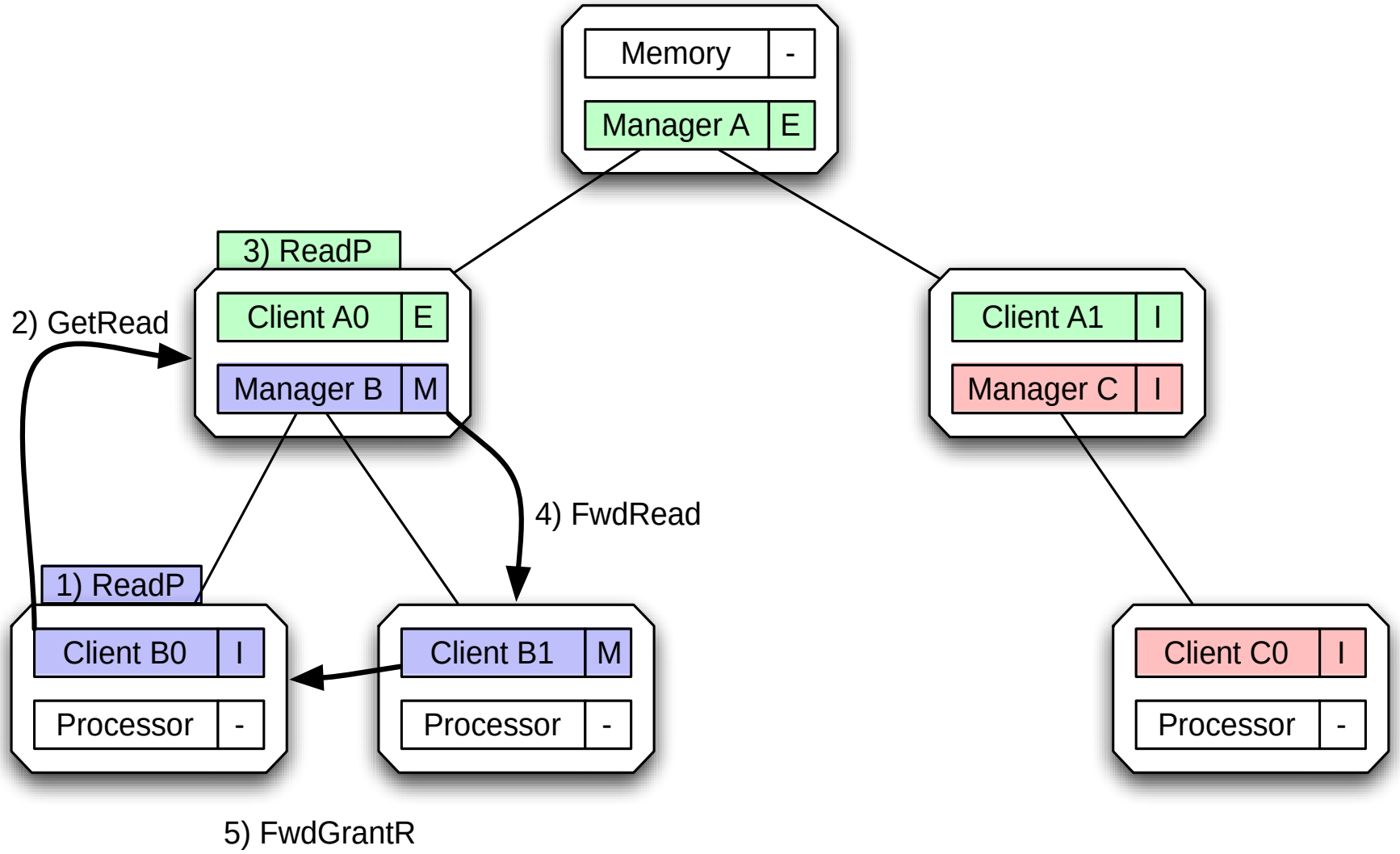
Special Functions

- Client: Eviction Permissions?
  - Evict_P and GetEvict
  - Why? What if in M/O state?
    - Directory is making assumptions about client's role
      - That client will fwd data to other caches
    - Client needs to inform directory before giving up ownership
- Manager: Downgrade (DwnInval and DwnRead)
  - Realm A has in the M state
  - Realm B asks for permission
    - Asks for write permission, A needs to become invalid
    - Asks for read permission, A only needs to give up exclusivity but can keep a copy

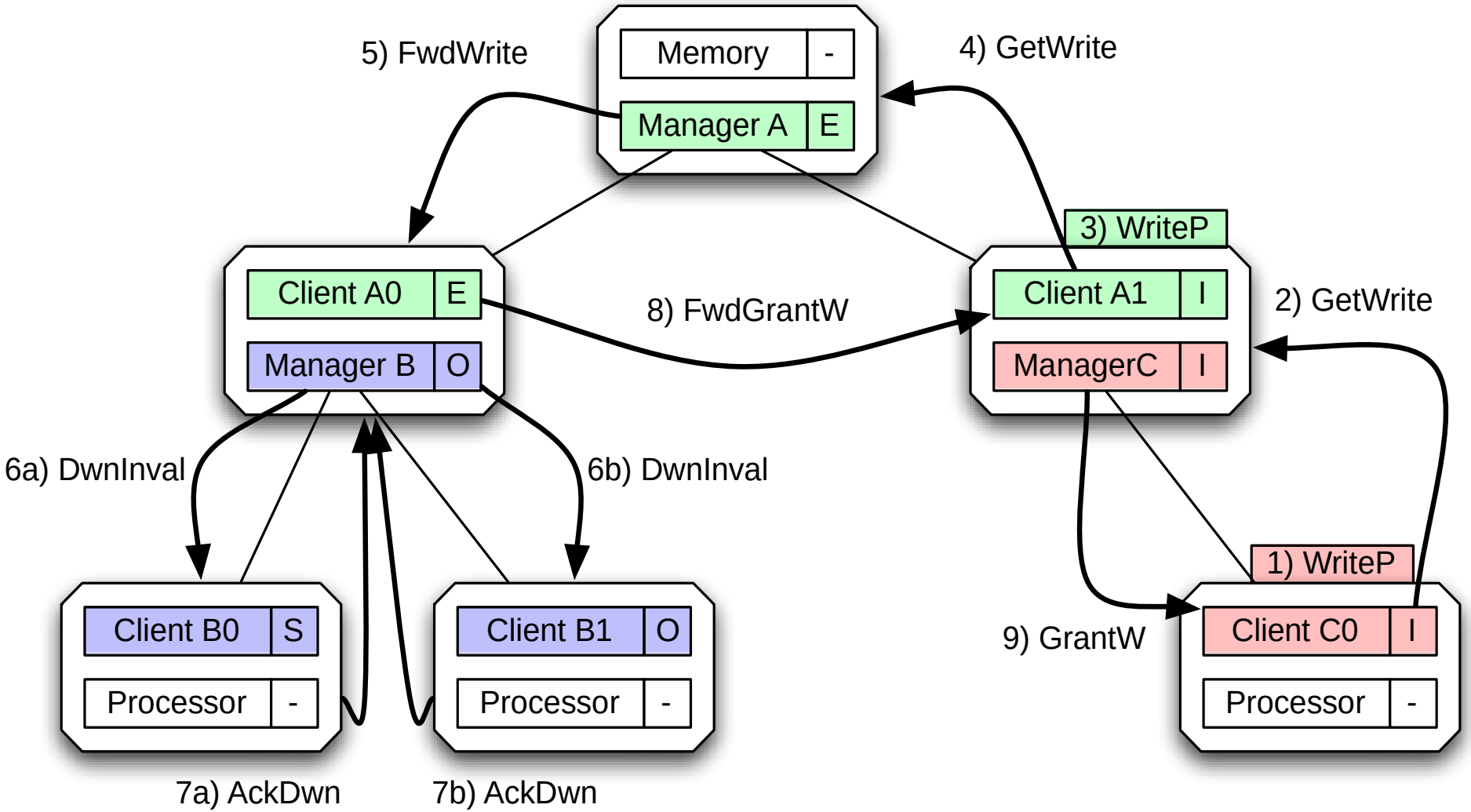
Permission Hierarchy Algorithm



Example – Realm Read Hit



Example – Realm Write Miss



Cache Model and Interface

A cache model has two interfaces:
1. The processor-cache interface and
2. The cache-network interface.

The Processor-Cache Interface

On the process-cache interface, the processor sends requests to the cache and the cache sends responds back.

Message from processor

The processor model's request is supposed to implement the following two functions:

1. ***get_addr().*** This function returns a 64-bit integer that is the memory address for which the cache request is made.
2. ***is_read().*** This function returns true if the request is a read(load), and false if it is a write(store).

The cache model's event handler for the processor-cache interface should be a templated function similar to the following:

```
template<typename T>
void my_cache_req_handler(int, T*);
```

The templated type T is the type of the request from the processor.

Message to processor

Currently it is required that the cache model sends back to the processor the same data type that it gets from the processor. Therefore, if the processor sends type T to the cache, then the cache must respond with the same type T.

Cache-Network Interface

If the cache model is not directly connected to the interconnection network, it can send/receive its own data type. If it is connected to the network, then it should send/receive **manifold::uarch::NetworkPacket**

The cache model's own data should be serialized and stored in the member variable (an array) data of **NetworkPacket**. The simplest way to do this is just using byte-wise copy:

```
MyDataType* obj;
NetworkPacket* pkt = new NetworkPacket();
*((MyDataType*)pkt->data) = obj;
```

A cache model could send two types of messages over the network:

1. Cache-to-cache messages, such as coherence messages.
2. Cache-to-memory messages.

Both are carried by NetworkPacket.

Cache-Network Interface….

A cache model also receives two types of messages: from another cache or from memory. The event handler for its network input should be a templated function as follows:

```
template<typename T>
void my_net_handler(int, manifold::uarch::NetworkPacket*);
```

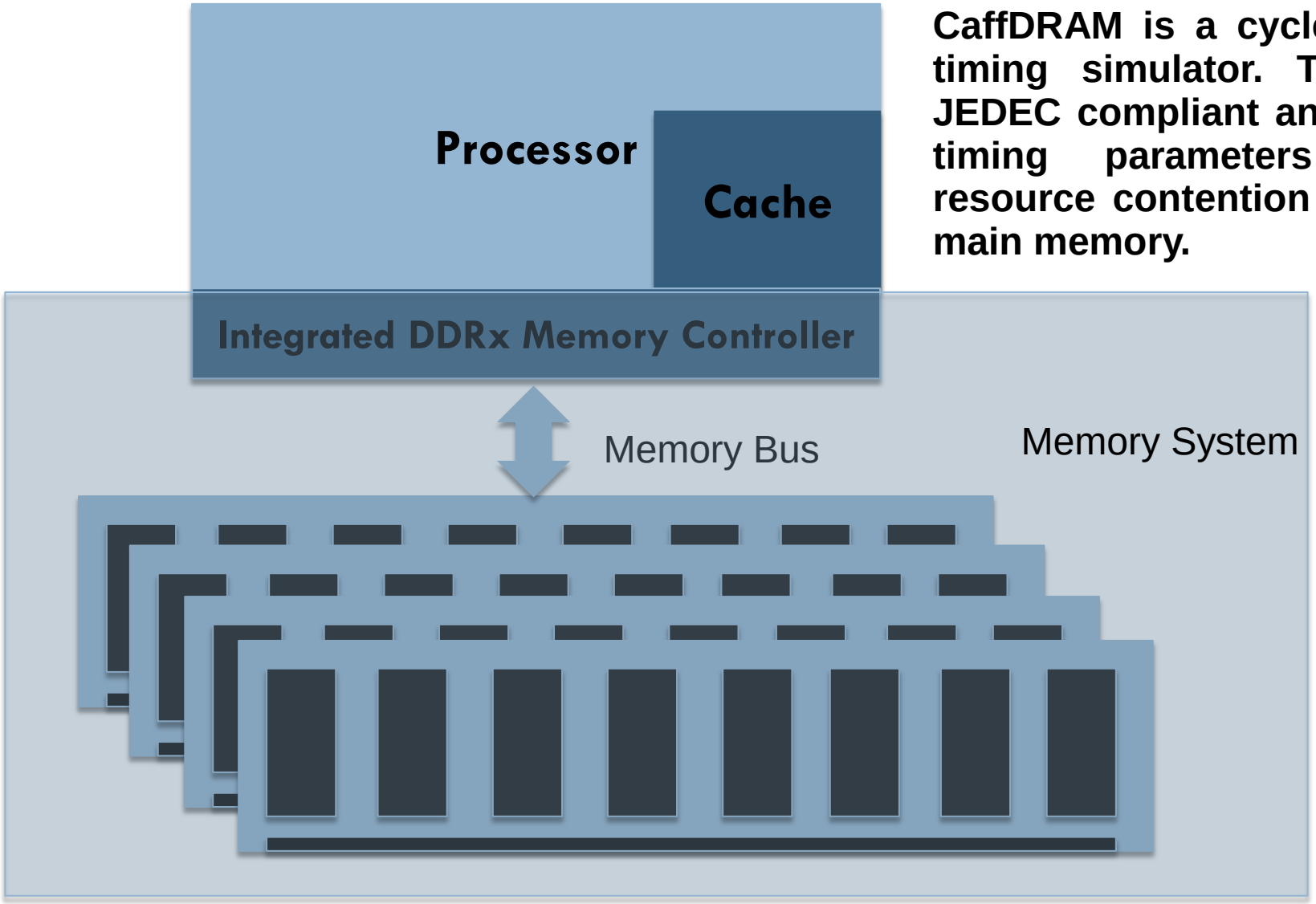
where the template parameter T is the data type from memory controller and is supposed to define the following two functions:

get_addr(). This functions returns a 64-bit integer that is the memory address for which the cache request is made.

is_read(). This function returns true if the request is a read(load), and false if it is a write(store). Pseudo code for the cache model's event handler for the cache-network interface is given below:

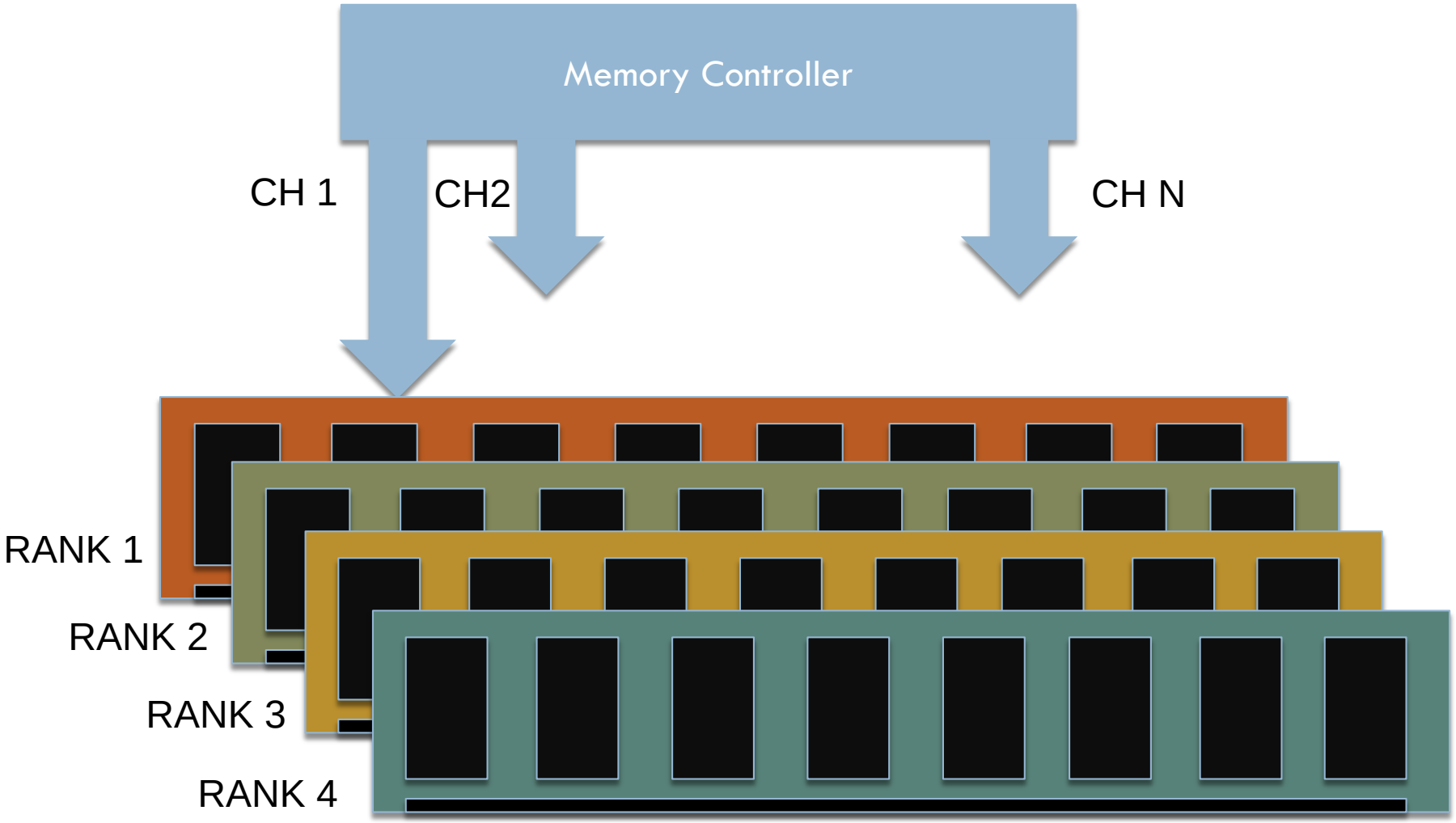
```
template<typename T>
void my_net_handler(int, manifold::uarch::NetworkPacket* pkt)
{
   IF pkt->type == coherence message THEN
      MyCohMsg* coh =  new MyCohMsg();
      *coh = *((MyCohMsg*)(pkt->data));
      process(coh);
   ELSE IF pkt->type == memory message THEN
      T objT = *((T*)(pkt->data));
      MyMemMsg* mem = new MyMemMsg;
      Set the member values of mem with objT;
      process(mem);
   END IF
}
```

CaffDRAM Overview



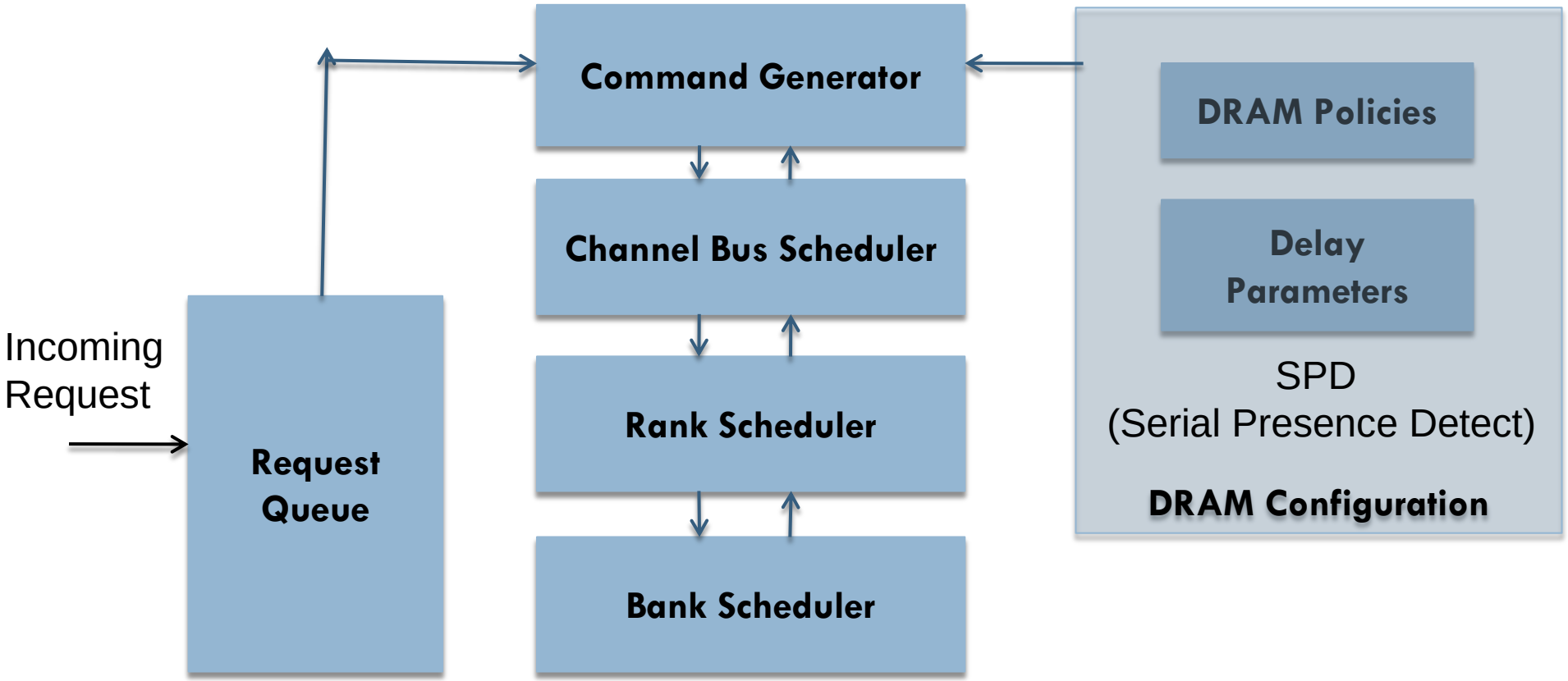
CaffDRAM is a cycle accurate DRAM timing simulator. This simulator is JEDEC compliant and models various timing parameters pertaining to resource contention within the DRAM main memory.

Memory Organization



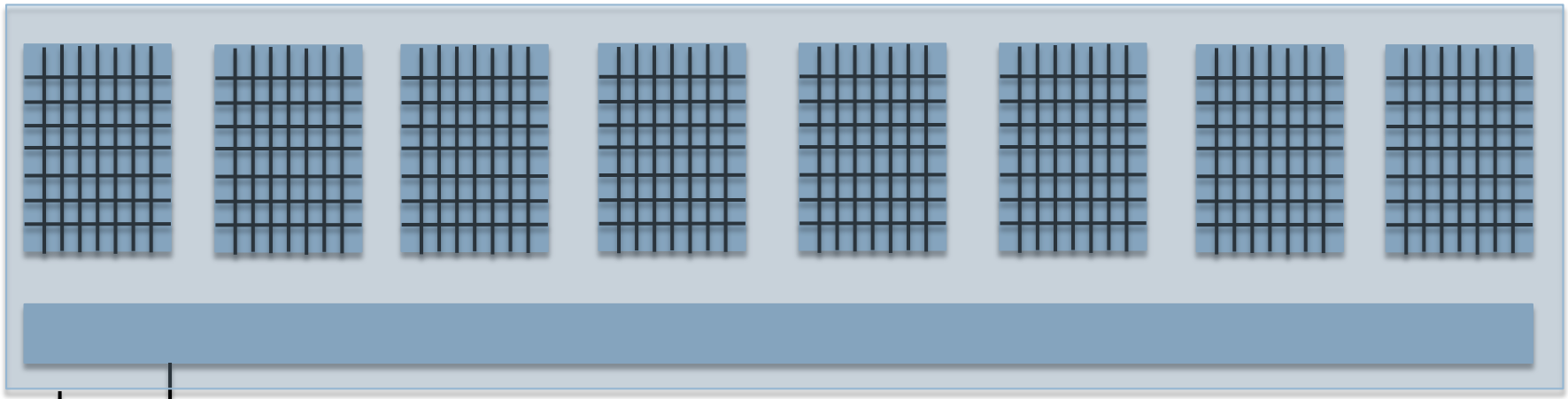
*A Single DIMM Module may consist of “one” or “two” Ranks depending upon its pin configuration. Each Side of a DIMM module is one “RANK” . A single “RANK” usually consists of 8 “x8” chips corresponding to a Data Bus width of 64 bits

DRAM Scheduling



Rank I/O Devices Contention Modeling

Rank Scheduler responsible for "I/O Device" resource contention



Rank I/O Device

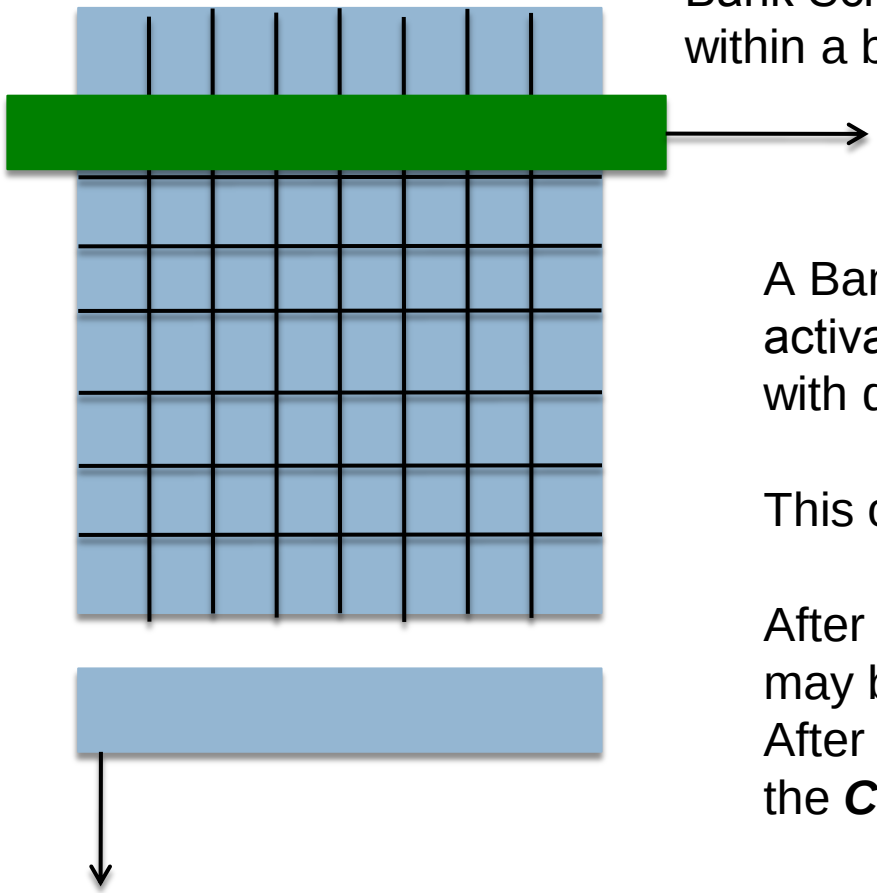
Single Rank

* On every "Read" multiple internal bursts are required to form a Longer burst on the data bus.
* On every "Write" data direction is reversed and these devices act as buffers for incoming data.

I/O Devices in use = Busy for "t_CCD" cycles

t_CCD = 2 beats for DDR = 1 memory cycle
t_CCD = 4 beats for DDR2 = 2 memory cycles
t_CCD = 8 beats for DDR3 = 4 memory cycles

Bank Contention Modeling



Bank Scheduler responsible for "Multiple Row Conflict" within a bank

An Active Row

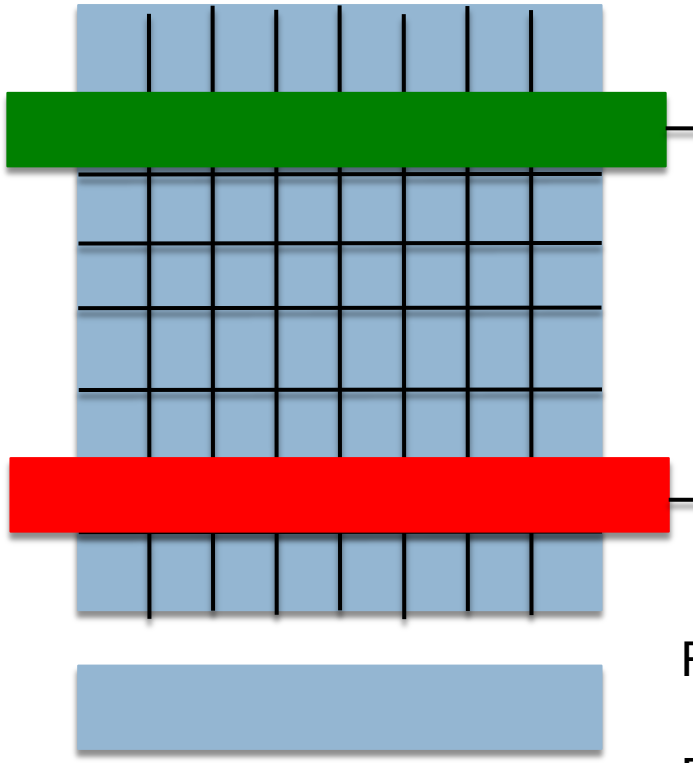
A Bank when accessed on a memory request activates a row which fills in the "sense amplifiers" with data from that row

This operation takes time = t_RCD

After time t_RCD, data from the "addressed columns" may be accessed.
After time = t_CAS (a.k.a. t_CL) data is placed on the ***Channel Data Bus***

Sense Amplifiers
(Each Bank has a set of sense Amplifiers shared by each row in a bank)

Bank Contention Modeling….



Previously Active Row

Newly accessed row within same bank
causing resource conflict

Row Activation = Bank busy for "t_RAS" cycles

Bank Precharge = Bank busy for "t_RP" cycles

*If a subsequent access is made to the same open row, the request can proceed immediately after the first without having to wait for t_RAS. In case of a different row the timing protocol must be followed

CaffDRAM Interface

* A memory controller has one interface: the memory controller-network interface. It sends/receives NetworkPacket which carries the memory requests and responses.

* The requests are defined in the cache model, therefore, the memory controller does not have the definition. For this reason, the event handler should be a template function as follows:

```
template<typename T>
void handler(int, manifold::uarch::NetworkPacket* pkt)
{
    T* request = (T*)(pkt->data);

    bool isRead = request->is_read();
    uint64_t addr = request->get_addr();
    ...
}
```

CaffDRAM Interface….

As can be seen, the request from cache is supposed to implement the following two functions:

get_addr(). This functions returns a 64-bit integer that is the memory address for which the cache request is made.
is_read(). This function returns true if the request is a read(load), and false if it is a write(store).

For response, the memory controller model can reuse the data type of the cache's requests, or it can define its own. In the latter case, the data type must also support the same two functions above.

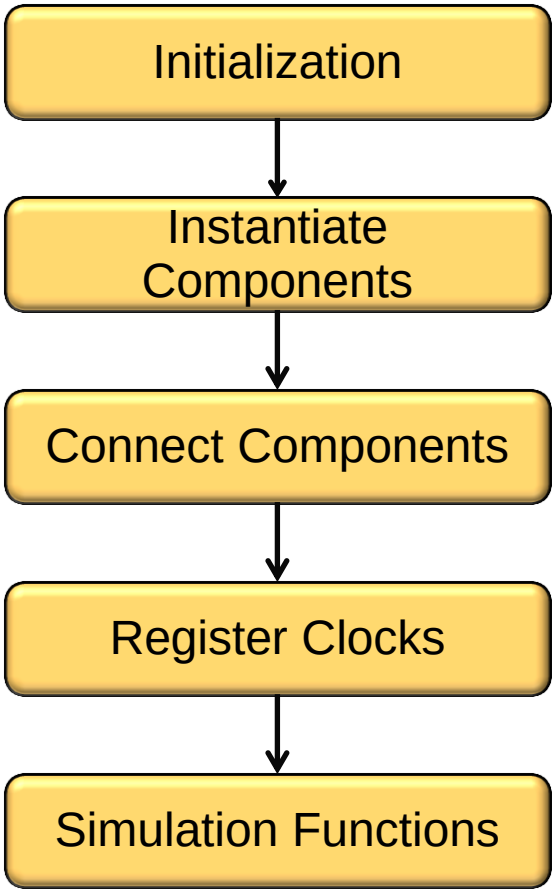
Message Types

The responses sent by the memory controller model use a message type (the type filed of NetworkPacket) that should be different from other message types. Therefore, the memory controller developer should not hard code the message type value. Instead, the type should be set in the constructor or an access function. The system model builder is responsible for setting the types.

Outline

- Introduction
- Execution Model and System Architecture
- Multicore Emulator Front-End
- Component Models
  - Cores
  - Network
  - Memory System
- Building and Running Manifold Simulations
- Physical Modeling: Energy Introspector
- Some Example Simulators

Building and Running Parallel Simulations



Configuration parameters

- From Manifold Library
- Inputs (trace, QSIM, etc.)

Instantiate Links

- Set Timing Behavior
- Time stepped vs. discrete event

Set Duration, Cleanup, etc.

Building and Running Parallel Simulations

- Kernel Interface
- Simulator Construction
- Logs and Statistics
- Demos

Kernel Interface

- Component functions
  - create component
    - component can have 0-4 constructor arguments
    - template allows constructor parameters to be any type
    - returns unique integer ID

```
//component-decl.h
template <typename T>
static CompId_t Create(LpId_t, CompName name=CompName("none"));

...

template <typename T, typename T1, typename T2, typename T3, typename T4>
static CompId_t Create(LpId_t, const T1&, const T2&, const T3&, const T4&,
CompName name=CompName("none"));
```

```
Component::Create<qsimclient_core_t>(lp, node_id, m_conf, cpuid, proc_settings);
```

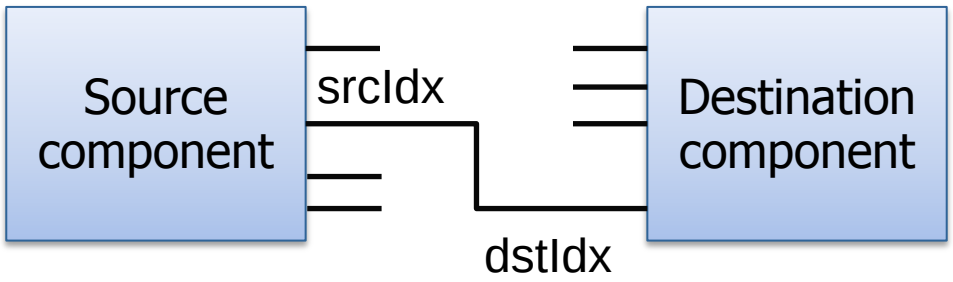
Kernel Interface

- Connect components
  - one-way connection

```
//manifold-decl.h
template<typename T, typename T2>
static void Connect(CompId_t srcComp, int srcIdx, CompId_t dstComp, int dstIdx, void
(T::*handler)(int, T2), Ticks_t latency);
```

  - two-way connection

```
//manifold-decl.h
template<typename T, typename T2, typename U, typename U2>
static void Connect(CompId_t comp1, int idx1, void (T::handler1)(int, T2), CompId_t
comp2, int idx2, void(U::*handler2)(int, U2), Clock& clk1, Clock& clk2, Ticks_t latency1,
Ticks_t latency2);
```



Kernel Interface

- Clock functions: constructor, Register()

```
//clock.h
Clock(double freq);

template<typename O>
static tickObjBase* Register(Clock& clk, O* obj, void (O::*rising)(void)
                                                 void (O::*falling)(void));
```

- simulation functions

```
//manifold-decl.h
static void Init(int argc, char**argv, SchedulerType=TICKED,
                 SyncAlg::SyncAlgType_t syncAlg=SyncAlg::SA_CMB_OPT_TICK,
                 Lookahead::LookaheadType_t la=Lookahead::LA_GLOBAL);

static void Finalize();

static void StopAt(Ticks_t stop);

static void Run();
```

Simulator Construction

- Steps for building a simulation program
  - Call Manifold::Init()
  - Build system model: Clock(); Create(), Connect(), Register()
  - Set simulation stop time: StopAt()
  - Call Manifold::Run()
  - Call Manifold::Finalize()
  - Print out statistics: print_stats()

Logs and Statistics

- Each component collects its own statistics
- A convention for printing stats is:
  - void print_stats(std::ostream&);

Example Simulators

- Simulator 1:
  - For demo purposes only
  - Builds a 2-core system
    - 2 Zesto cores
    - MCP cache
    - Iris(2x2 torus)
    - CaffDRAM
  - Runs sequential or parallel (3 LPs) simulation

- Simulator 2:
  - Part of software distribution
  - 3 programs: work with Qsim server, Qsim lib, and traces, respectively
  - Core model can be replaced with one-line change to configure file

Sample Results: Setup

- 16, 32, 64-core CMP models
- 2, 4, 8 memory controllers, respectively
- 5x4, 6x6, 9x8 torus, respectively
- Host: Linux cluster; each node has 2 Intel Xeon X5670 6-core CPUs with 24 h/w threads
- 13, 22, 40 h/w threads used by the simulator on 1, 2, 3 nodes, respectively
- 200 Million simulated cycles in region of interest (ROI)
  - Saved boot state and fast forward to ROI

Sample Results: Simulation Time in Minutes

	16-core		32-core		64-core	
	Seq.	Para.	Seq.	Para.	Seq.	Para.
dedup	1095.7	251.4 (4.4X)	2134.8	301.3 (7.1X)	2322.9	345.3 (6.7X)
facesim	1259.3	234.9 (5.4X)	2614.2	303.6 (8.6X)	3170.2	342.3 (9.3X)
ferret	1124.8	227.8 (4.9X)	1777.9	255.6 (7.0X)	2534.3	331.3 (7.6X)
freqmine	1203.3	218.0 (5.5X)	1635.6	245.6 (6.7X)	2718.9	337.3 (8.1X)
stream	1183.8	222.7 (5.3X)	1710.6	244.3 (7.0X)	4796.4	396.2 (12.1X)
vips	1167.0	227.3 (5.1X)	1716.3	257.2 (6.7X)	2564.6	337.9 (7.6X)
barnes	1039.9	224.3 (4.6X)	1693.0	283.3 (6.0X)	3791.8	341.4 (11.1X)
cholesky	1182.4	227.2 (5.2X)	1600.3	245.7 (6.5X)	4278.3	402.1 (10.6X)
fmm	1146.3	229.6 (5.0X)	1689.8	253.6 (6.7X)	5037.2	416.1 (12.1X)
lu	871.2	156.4 (5.6X)	1475.8	204.6 (7.2X)	4540.3	402.7 (11.3X)
radiosity	1022.3	228.8 (4.5X)	1567.5	250.4 (6.3X)	2813.5	350.3 (8.0X)
water	671.5	158.4 (4.2X)	1397.3	236.7 (5.9X)	2560.1	356.3 (7.2X)

Sample Results: Simulation in KIPS

	16-core		32-core		64-core	
	Seq.	Para.	Seq.	Para.	Seq.	Para.
dedup	49.28	211.98	48.56	340.61	16.88	136.82
facesim	58.66	316.42	47.90	401.72	30.01	278.94
ferret	57.77	284.81	35.41	239.59	18.10	139.54
freqmine	57.15	314.99	37.01	248.60	19.37	140.18
stream	58.77	314.34	36.73	260.04	41.42	419.77
vips	58.03	298.09	34.92	236.06	18.02	131.02
barnes	30.66	151.32	32.62	168.74	39.61	219.63
cholesky	57.95	301.47	38.87	254.54	45.68	491.46
fmm	51.01	252.46	37.90	255.86	40.94	525.51
lu	50.93	155.74	39.78	119.46	46.23	485.04
radiosity	50.87	206.96	53.15	229.02	36.29	268.70
water	27.86	95.81	29.85	132.22	25.72	179.88
Mean	50.75	242.03	39.39	240.54	31.52	284.71
Median	54.08	268.63	37.45	244.10	33.15	244.16

Sample Results: KIPS per Hardware Thread

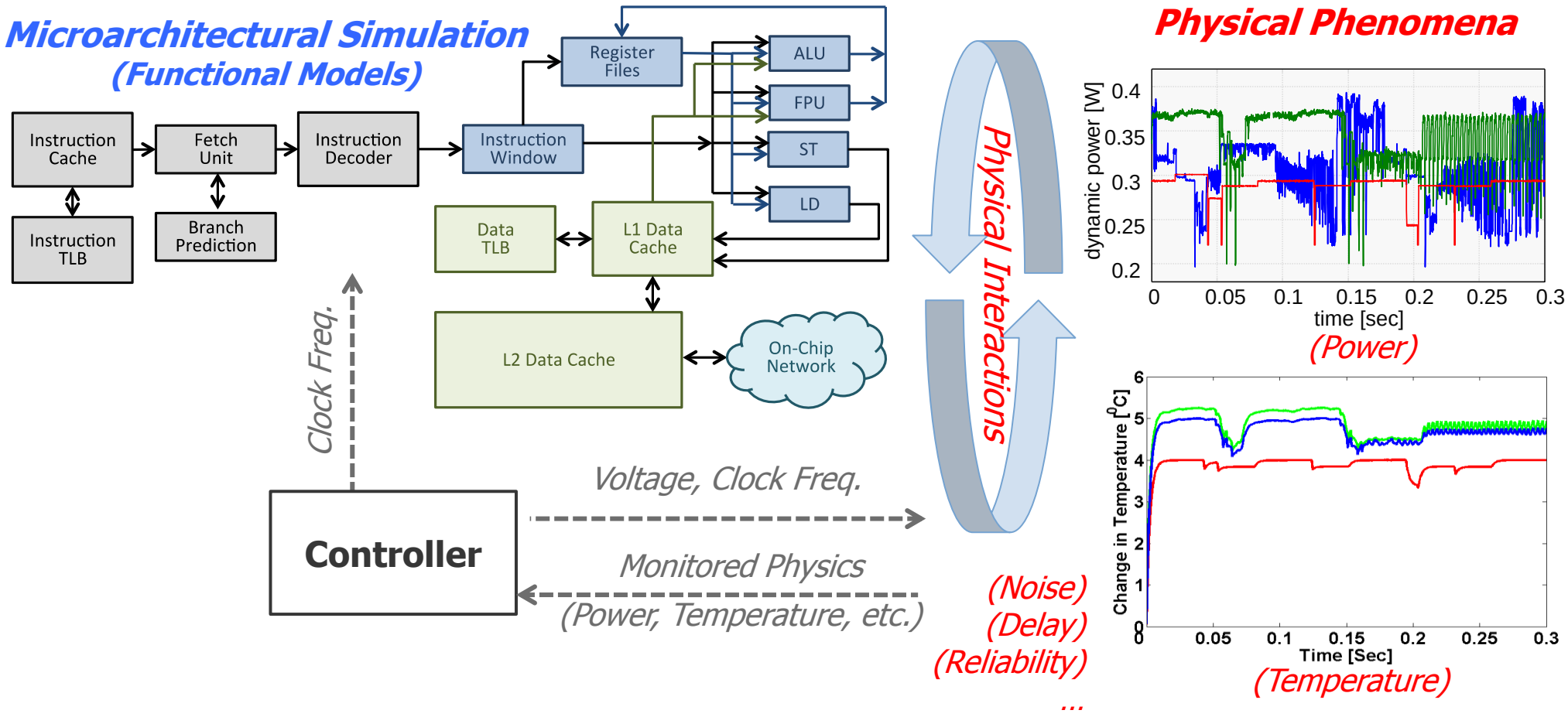
	16-core		32-core		64-core	
	Seq.	Para.	Seq.	Para.	Seq.	Para.
dedup	49.28	16.31	48.56	15.48	16.88	3.42
facesim	58.66	24.34	47.90	18.26	30.01	6.97
ferret	57.77	21.91	35.41	10.89	18.10	3.49
freqmine	57.15	24.23	37.01	11.30	19.37	3.50
stream	58.77	24.18	36.73	11.82	41.42	10.49
vips	58.03	22.93	34.92	10.73	18.02	3.28
barnes	30.66	11.64	32.62	7.67	39.61	5.49
cholesky	57.95	23.19	38.87	11.57	45.68	12.29
fmm	51.01	19.42	37.90	11.63	40.94	13.14
lu	50.93	11.98	39.78	5.43	46.23	12.13
radiosity	50.87	15.92	53.15	10.41	36.29	6.72
water	27.86	7.37	29.85	6.01	25.72	4.50
Mean	50.75	18.62	39.39	10.93	31.52	7.12
Median	54.08	20.66	37.45	11.09	33.15	6.10

Outline

- Introduction
- Execution Model and System Architecture
- Multicore Emulator Front-End
- Component Models
  - Cores
  - Network
  - Memory System
- Building and Running Manifold Simulations
- Physical Modeling: Energy Introspector
- Some Example Simulators

- Energy Introspector: Integration of Physical Models

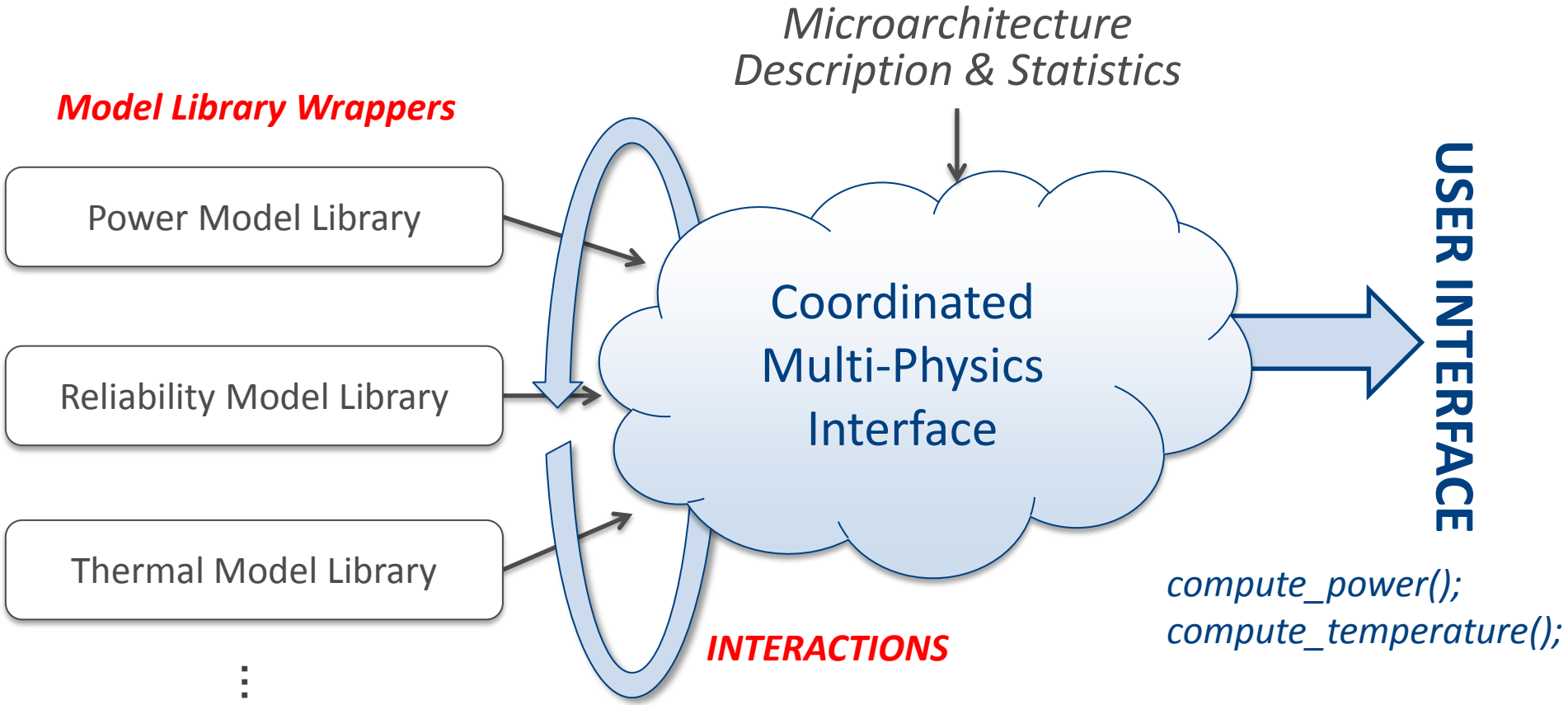
Architecture-Physics Co-Simulation



- Architecture Simulations ➔ Physical phenomena
- Physical Phenomena ➔ Control Algorithms
- Control Algorithms ➔ Architectural Executions

Introduction to Energy Introspector

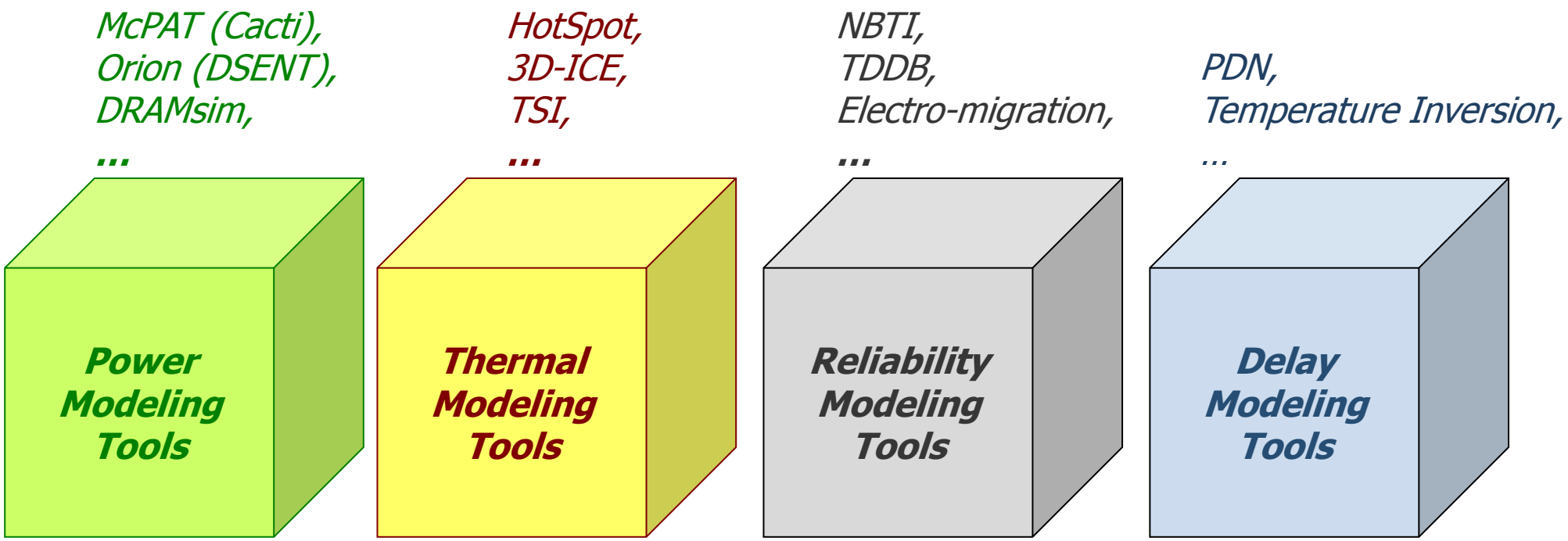
- *Energy Introspector* (**EI**) is a simulation framework to facilitate the (selective) uses of different models and capture the interactions among microprocessor physics models.



Available at *www.manifold.gatech.edu*

Multi-Physics / Multi-Model Modeling

- *Different physical properties* are modeled with *different modeling tools*.



- Need to be able to easily integrate new models as they become available
  - Costly when tightly integrated into microarchitecture simulation models

Physical Interaction Interface

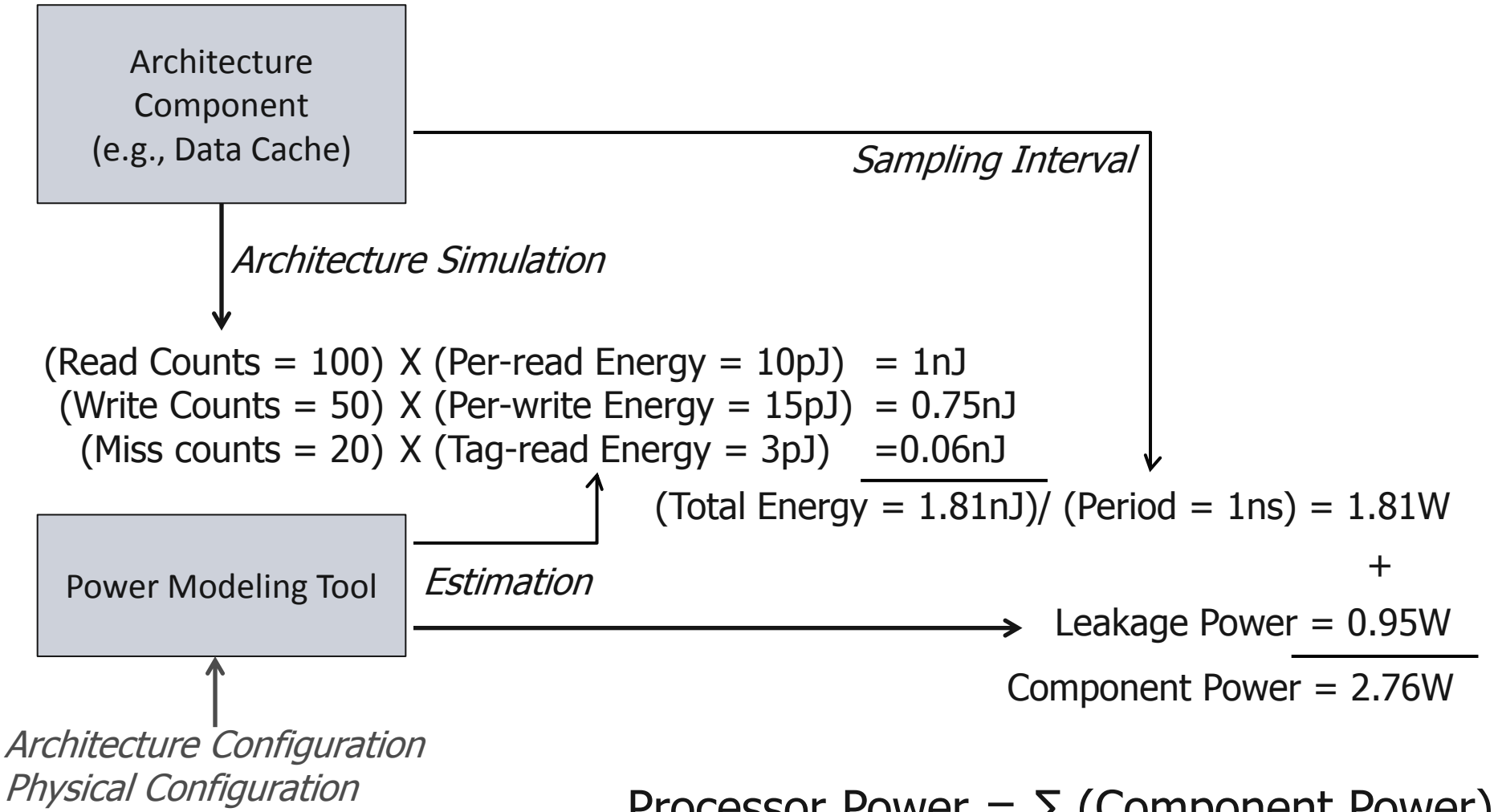
- Goals:
  1. **Coordinated Modeling**: The interface supports the modeling of *different physical phenomena* in *the same domain of analysis* by capturing their *interactions*; vs. trace-driven offline analysis
  2. **Standardization**: *Different tools* are integrated in the compatible way.
     - *Data definition* is required for common, shared data.
     - *Different tools of the same data* type can used in the identical way.
  3. **Configurable Interface**: Due to the integration of different tools, there must be *a method to represent the different levels of processor components*.
     - Package | Floor-planning | Architecture/Circuit-level Blocks
  4. **Data Synchronization:** *Data are shared* between multiple tools.
  5. **User Interface**: User API is provided to trigger the calculation.
     - The interface handles the *data synchronization* and *interactions*.
- We introduce the **Energy Introspector (EI)** interface.

COORDINATED MODELING:

<*Multiple physical phenomena modeled in the same domain of analysis*>

Energy | Power Modeling

- Energy (or power) is characterized w.r.t. *architectural activities*.

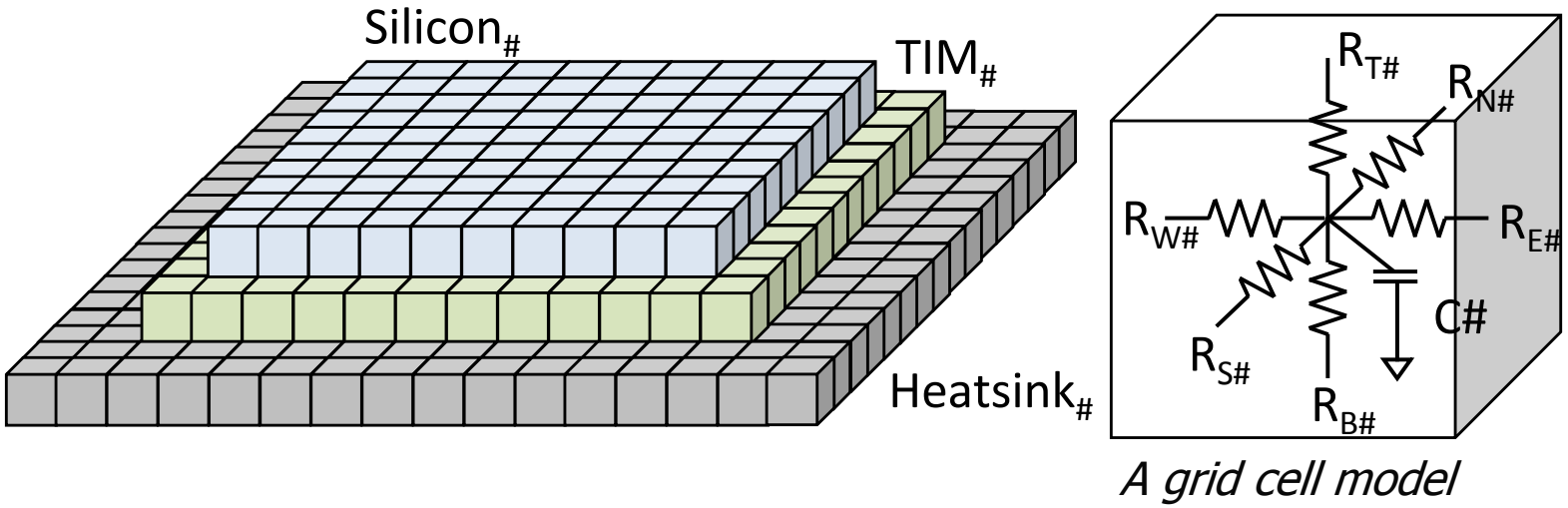


** Numbers are randomly chosen to show an example.*

Thermal Modeling

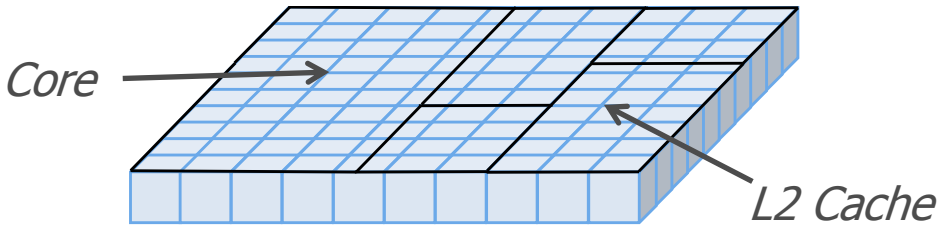
- Temperature is characterized at the *package-level* w.r.t spatiotemporal *distribution of power* (i.e., power density).

The package is represented with layers of thermal grid cells.



A grid cell model

Architectural components are placed on the processor die (*floor-planning*).



** Floor-plan is arbitrarily drawn to show an example.*

Reliability (Hard Failure) Modeling

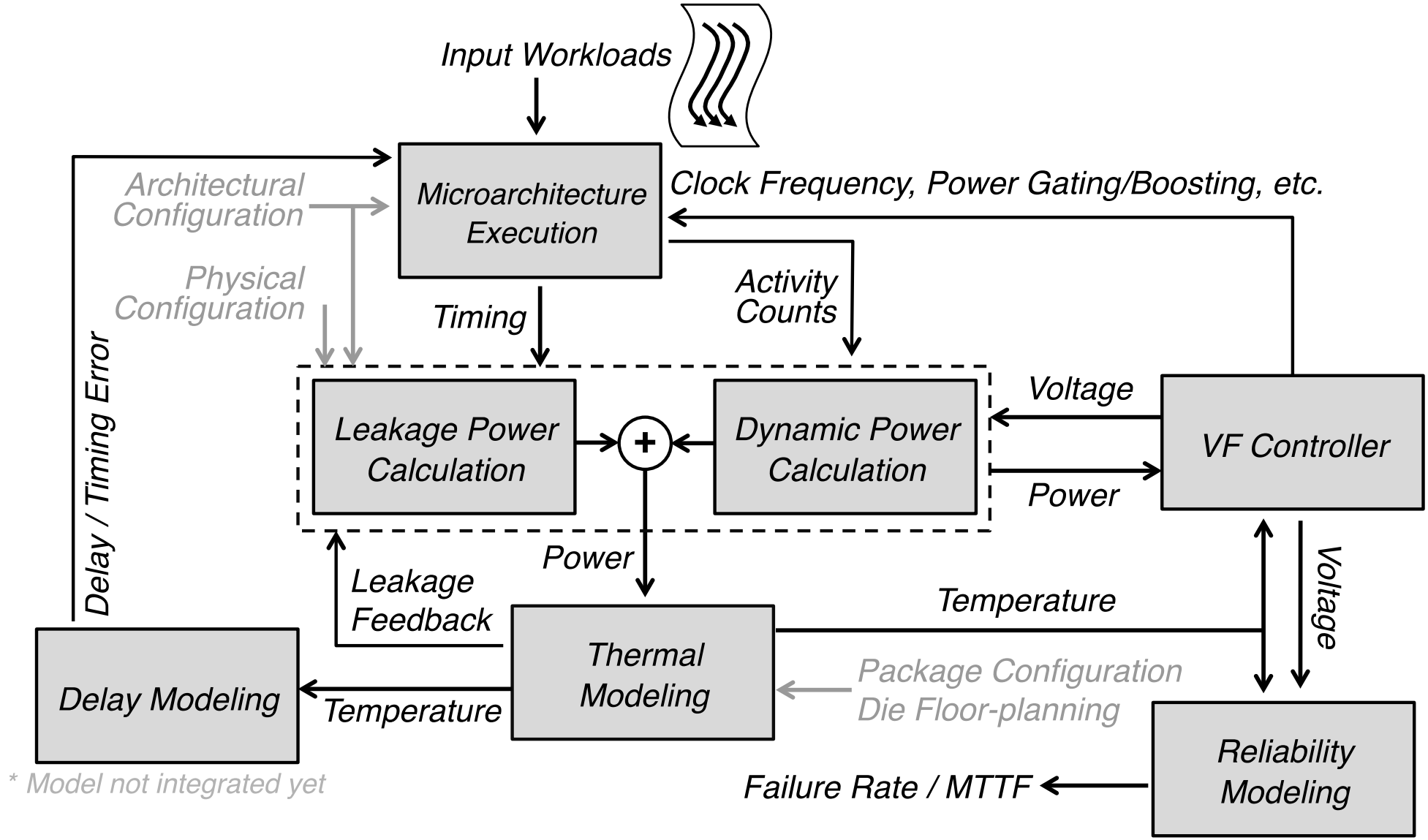
Failures	Description
Electro-migration (EM), λ_{EM}	Directional transport of electrons and metal atoms in interconnect wires causes degradation and failure.
Time dependent dielectric breakdown (TDDB), λ_{TDDB}	Wearout of gate oxide caused by continued application of electric field leads to short between gate and substrate.
Hot carrier injection (HCI), λ_{HCI}	Electrons with sufficient kinetic energy overcomes the barrier to gate oxide and cause degradation.
Bias temperature instability (NBTI/PBTI), λ_{BTI}	Gradual degradation causes threshold voltage shift and eventually timing errors.
Stress migration (SM), λ_{SM}	Differences in the expansion rates of metals cause mechanical stress.
Thermal cycling (TC), λ_{TC}	Temperature cycle accumulates fatigue to materials.

Failure Rate = $\sum (\lambda_{FAILURE})$ Mean-time-to-failure (MTTF)= 1/Failure Rate

The failure rate is a function of operating conditions (i.e., voltage, temperature).

Architecture-Level Physical Modeling

Abstract Representation of Architecture-Physics Interactions



STANDARDIZATION:

<Compatible Integration of Different Modeling Tools>

Data Definition

- Different models use different types and units of data.
- The EI defines *the modeled physical data*.

```
// Basic Units
typedef uint64_t Count;
typedef double Meter;
typedef double Hertz;
typedef double Volt;
typedef double Kelvin;
typedef double Watt;
typedef double Joule;
typedef double Unitless;
typedef double Second;
```

```
// Activity Counters
class counter_t {
  Count switching;
  Count read;
  Count write;
  Count read_tag;
  Count write_tag;
  Count search;
  void clear();
};
```

```
// Energy
class energy_t {
  Joule total;
  Joule dynamic;
  Joule leakage;
  void clear();
  Joule get_total();
};

// Power
class energy_t {
  Watt total;
  Watt dynamic;
  Watt leakage;
  void clear();
  Watt get_total();
};

power_t operator*(const energy_t &e,
const Hertz &f);
power_t operator/(const energy_t &e,
const Second &s);
```

```
// Dimension
class dimension_t {
  Index layer_index;
  char layer_name[LEN];
  Meter left, bottom;
  Meter width, height;
  MeterSquare area;
  void clear();
  MeterSquare get_area();
};
```

```
// Grid
template <typename T>
class grid_t {
  unsigned x, y, z;
  unsigned count;
  std::vector<T> element;
  T& operator[](unsigned i)
...
};
```

Integration of Modeling Tools (1)

- Tools that model the same physical phenomena are categorized into the same *library*.
- Each library defines the *functions* to be provided by the models.
  - **Energy Library**:
    - It calculates *runtime energy/power* dissipation w.r.t. architectural activity
    - It estimates *TDP power* and *area* (if possible).
    - Integrated models: *McPAT*, *IntSim*, to be integrated: *DSENT*, *DRAMsim*
  - **Thermal Library**:
    - It calculates *transient/steady-state temperature* w.r.t. power inputs.
    - It requires *floor-planning* for the functional dies.
    - It provides different levels of *thermal data*; grid, block, or point temp.
    - Integrated models: *HotSpot*, *3D-ICE*, to be integrated: *TSI*, *Microfluidics*
  - **Reliability Library**:
    - It calculates *failure rate* w.r.t. operation conditions (i.e., voltage, temp).
    - Integrated model: *Hard Failure*

Integration of Modeling Tools (2)

```
// Base Library Class
class model_library_t {
  virtual ~model_library_t();
  virtual bool update_library_variable(int Type, void *Value, bool IsLibraryVariable)=0;
  virtual void initialize()=0;
};
```

```
// Energy Library Class
class energy_library_t : public model_library_t {
  virtual ~energy_library_t();
  virtual unit_energy_t get_unit_energy()=0;
  virtual power_t get_tdp_power(Kelvin MaxTemperature)=0;
  virtual power_t get_runtime_power(Second Time, Second Period, Counter_t Counter)=0;
  virtual MeterSquare get_area()=0;
};
```

```
// Thermal Library Class
class thermal_library_t : public model_library_t {
  virtual ~thermal_library_t();
  virtual void calculate_temperature(Second Time, Second Period)=0;
  virtual grid_t<Kelvin> get_thermal_grid()=0;
  virtual Kelvin get_partition_temperature(Comp_ID ComponentID, int MappingType)=0;
  virtual Kelvin get_point_temperature(Meter X, Meter Y, Index Layer)=0;
  virtual void push_partition_power(Comp_ID ComponentID, power_t PartitionPower)=0;
  virtual void add_pseudo_partition(std::string ComponentName, Comp_ID ComponentID)=0;
...
};
```

Integration of Modeling Tools – Example

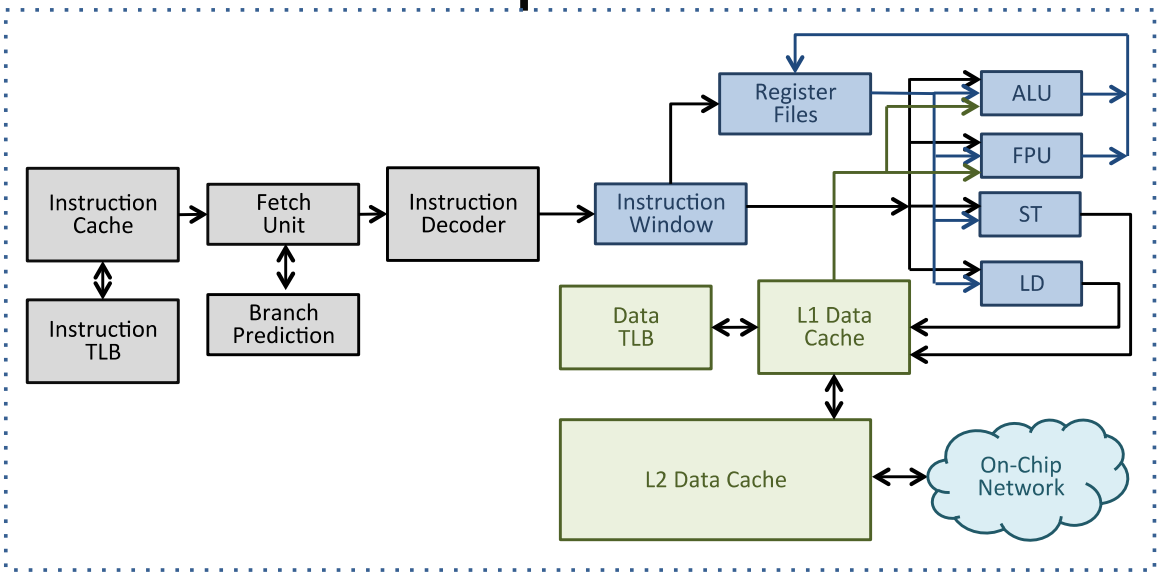
- Each integrated tool becomes a *subclass* of one of the model libraries.

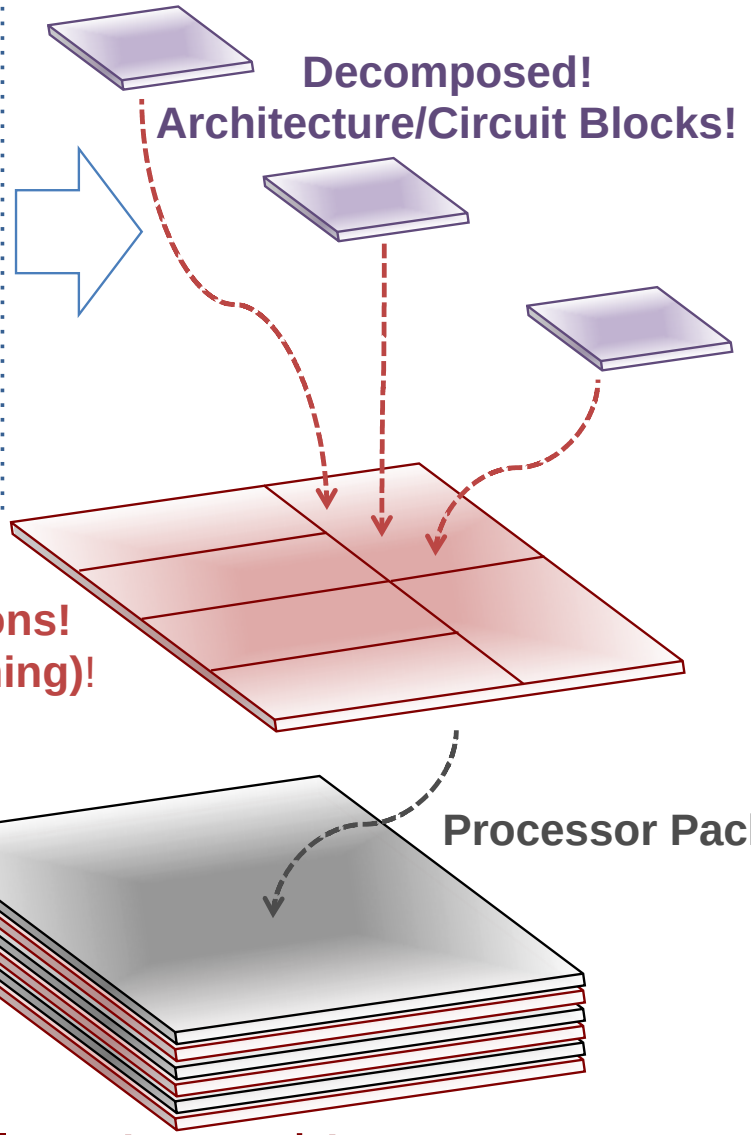
```
// Example: Integration of HotSpot
#include <HotSpot header files>

class thermallib_hotspot : public thermal_library_t {
  ~thermalib_hotspot();
  void initialize() {
    /* DEFINE A METHOD TO CREATE AND INITIALIZE HOTSPOT DATA STRUCTURES AND VARIABLES */
  }
  void update_library_variable(int Type, void *Value, bool IsLibraryVariable) {
    /* DEFINE A METHOD TO DYNAMICALLY UPDATE HOTSPOT DATA STRUCTURES AND VARIABLES */
  }
  void calculate_temperature(Second Time, Second Period) {
    /* DEFINE A METHOD TO ACCESS HOTSPOT DATA STRUCTURES AND CALCULATE TEMPERATURE */
  }
  void push_partition_power(Comp_ID ComponentID, power_t PartitionPower) {
    /* UPDATE THE PARTITION (i.e., FLOOOR-PLAN) POWER FOR THE NEXT TEMP. CALCULATION */
  }
  Kelvin get_partition_temperature(Comp_ID ComponentID, int MappingType) {
    /* RETRIEVE THE PARTITION (i.e., FLOOR-PLAN) TEMP. AFTER calculate_temperature(); */
  }
...
};
```

CONFIGURABLE INTERFACE:

<A method to represent the different levels of processor components – from processor package to microarchitecture units>

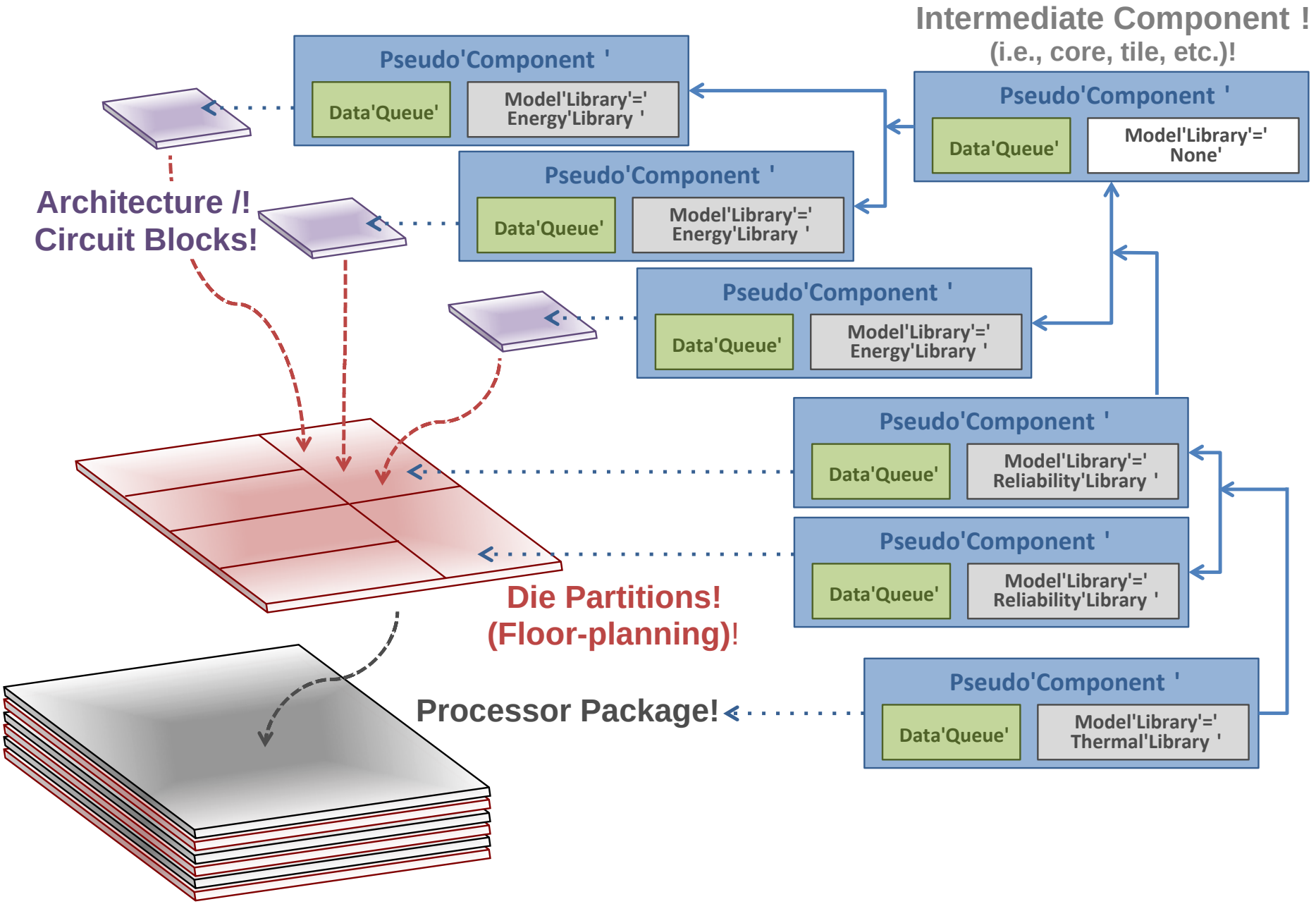
Abstract Representation of Processor Hierarchy





How can we flexibly mix and match models, microarchitecture components, and physical geometry?

Processor Representation (1)



Processor Representation (2)

- *Different physical phenomena* are characterized and *different levels of processor components*.
- The EI is comprised of *pseudo components* that can represent any processor components
  - A pseudo component can represent a *package*, *die*, *core*, *floor-plan block*, etc.
  - A pseudo component can be *associated with an integrated modeling tool* to *characterize a physical phenomenon*.

```
// Pseudo Component
class pseudo_component_t {
  pseudo_component_t(Comp_ID ID, libEI::energy_introspector_t *EI, std::string Name,
                     libconfig::Setting *ConfigSetting, bool IsRemote);
  Comp_ID id; // Integer ID
  int rank; // MPI Rank (for parallel simulation)
  std::string name; // String name
  model_library_t *model_library; // Associated library tool
  queue_t queue; // Data queue that stores computed results or shared data
  libEI::energy_introspector_t *energy_introspector; // Pointer to main interface
  libconfig::Setting *setting; // libconfig setting for this pseudo component
  libEI::pseudo_component_t *superset; // A parent component in the processor hierarchy
  std::vector<libEI::pseudo_component_t*> subset; // Child components
...
};
```

DATA SYNCHRONIZATION:

<Time synchronization and data sharing between library models>

Data Synchronization (1)

```
for(all components)
    EI->calculate_power(ComponentID,Time,Period,Counter,/*IsTDP*/false);

EI->synchronize_data(packageID,Time,Period,EI_DATA_POWER); // Sync Power
EI->calculate_temperature(PackageID,Time,Period,/*PowerSync*/false);
```

Or, alternatively

```
for(all components)
    EI->calculate_power(ComponentID,Time,Period,Counter,/*IsTDP*/false);

EI->calculate_temperature(PackageID,Time,Period,/*PowerSync*/true);
```

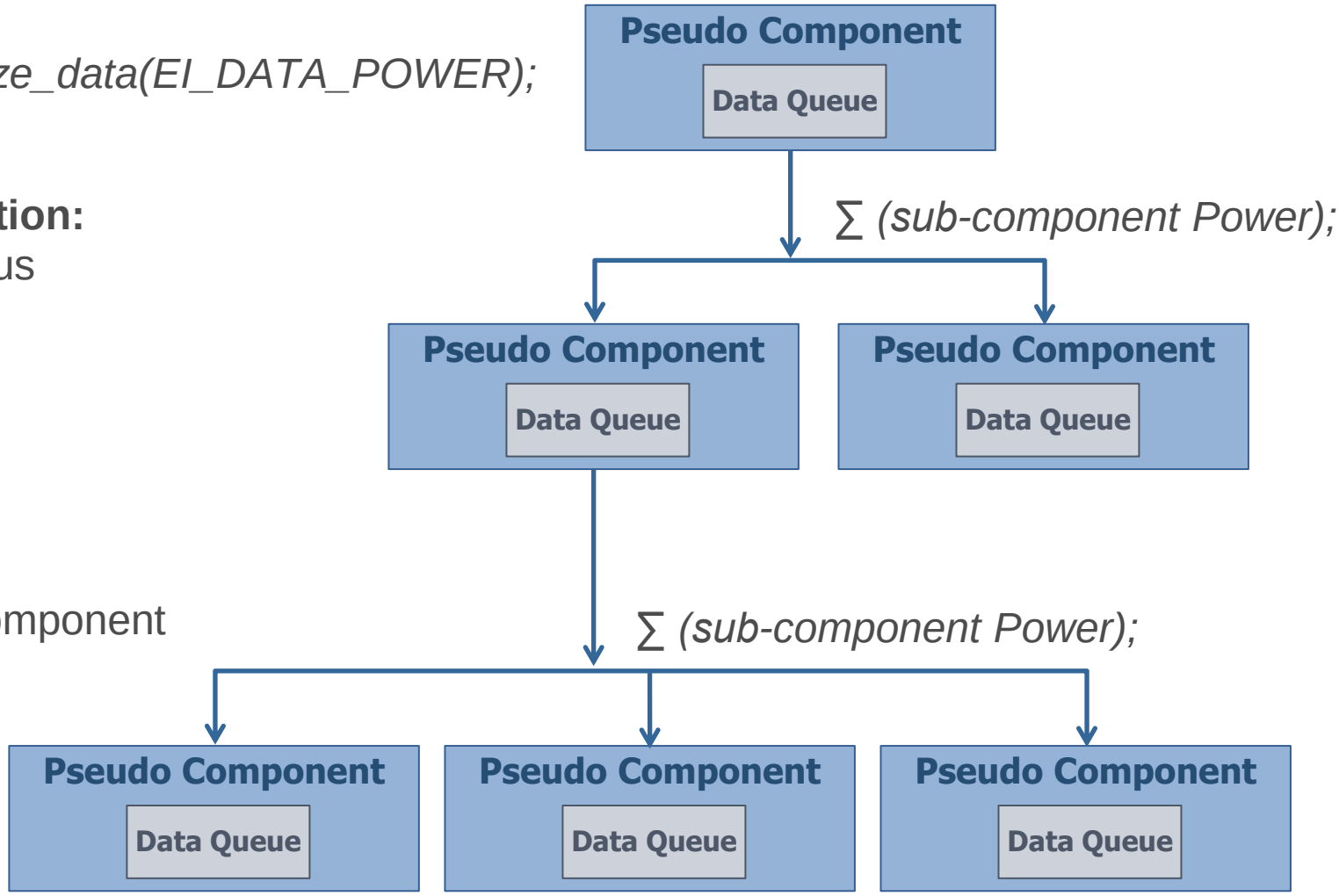
- The EI provides handles *data synchronization* across *pseudo components*.
- Different data types have different synchronization method:
  - **Power, Area**: *added up* from the bottom of the pseudo component tree.
  - **Temperature, Voltage, Clock Freq**: *applied identically* to sub-components.

Data Synchronization (2)

synchronize_data(EI_DATA_POWER);

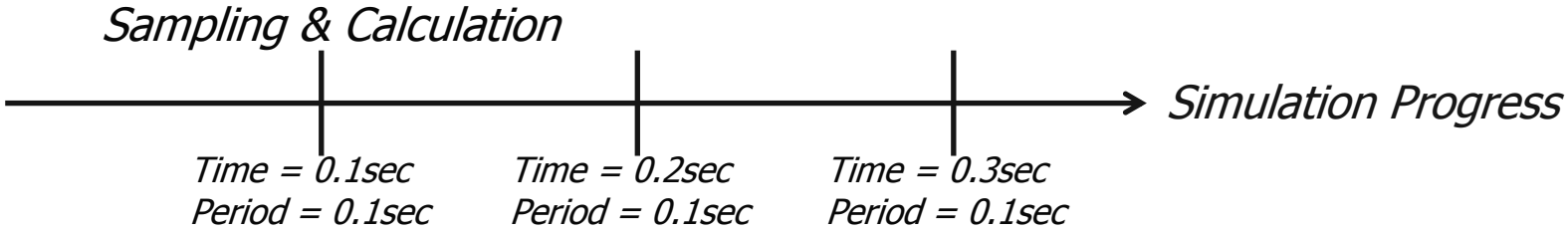
Queue Error Detection:

- Time discontinuous
- Time outorder
- Time overlap
- Time invalid
- Data duplicated
- Data invalid
- Data type invalid
- Invalid pseudo component



Data Manipulation

- Each pseudo component includes *data queues* to store *computed results* or *shared data*.
- Data are of discrete time, and thus associated with *time*, *period* information.



  - **Closed Queue**:
    - This queue type is used for *periodically calculated and sampled data* such as power, temperature, etc.
    - Data are *continuous to the left*; valid interval = (time-period, time]
  - **Open Queue**: This queue type is used for *aperiodically controlled data* such as voltage, clock frequency, etc.
    - Data are *continuous to the right*; valid interval = [time, unknown)
- The insertion of data triggers the *callback function* of the library model (if any) to *update dependent variables and states*.

Data Queue Structure

- Each pseudo component includes *data queues* to store *computed results* or *shared data*.

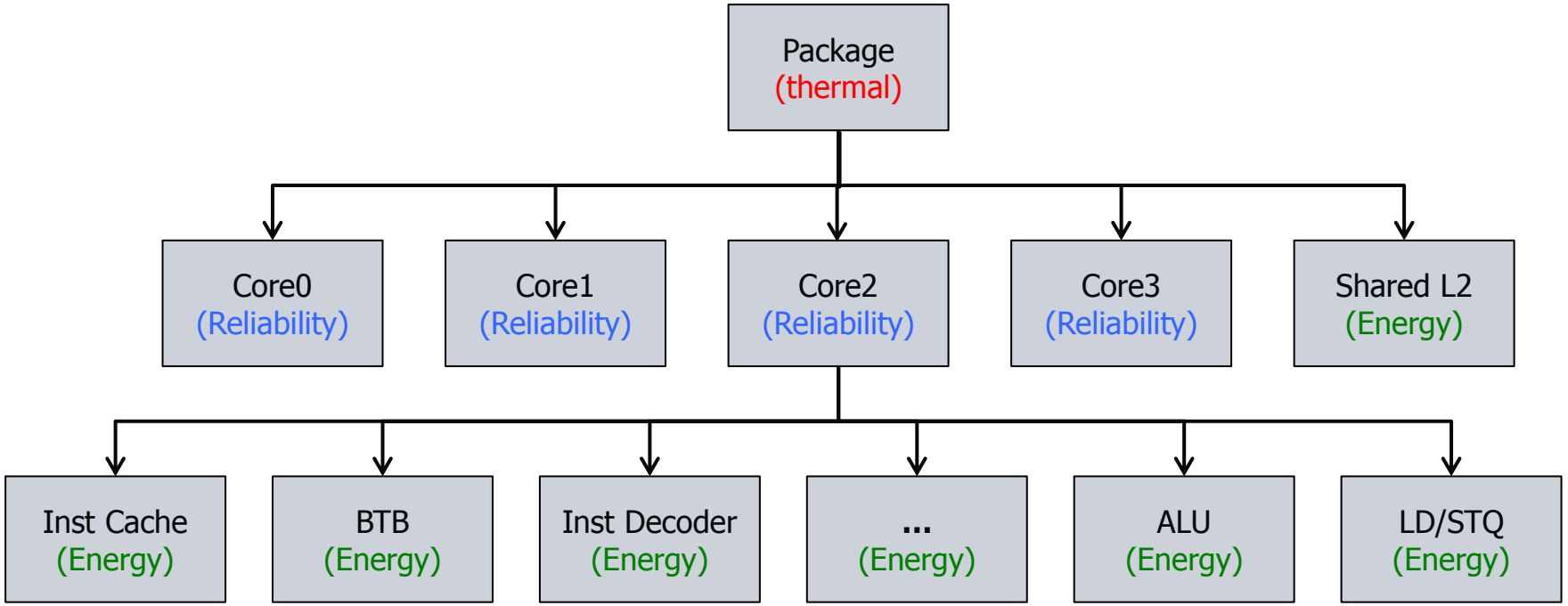
```
// Base Queue
class base_queue_t {
  virtual ~base_queue_t();
  void reset();
  int get_error();
  unsigned size; // Queue length
  int error; // Error code
};
```

```
// Individual Data Queue
template <typename T>
class data_queue_t : public base_queue_t{
  ~data_queue_t();
  void reset();
  void push(Second Time, Second Period,
            T Data);
  void overwrite(Second T, Second P, T D);
  T pull(Second T, Second P);
  int queue_type; // Open / Close Queue
  int data_type; // Data Type
  std::map<data_time_t,T> queue;
};
```

```
// Pseudo Component Data Queue
Class queue_t {
  ~data_queue_t();
  template <typename T>
  void create(unsigned QueueLength,
              int DataType, int QueueType);
  template <typename T>
  void push_back(Second Time, Second Period,
                 int DataType, T Data);
  template <typename T>
  void overwrite(Second Time, Second Period,
                 int DataType, T Data);
  template <typename T>
  T get(Second Time, Second Period,
        int DataType);
  void register_callback(model_library_t *Lib,
  bool(model_library_t::*update_library_variable)
  (int,void*,bool));
  void callback(int DataType, void *Data);
  int get_error();
  void reset();
  std::map<int,base_queue_t*> queue;
};
```

INPUT CONFIG & USER INTERFACE:

<*EI Configuration* and *User API*>



** Within () shows the modeled library at each pseudo component.*

Thermal Library = 3D-ICE

Reliability Library = Hard Failures

Energy Library = McPAT

Input Configuration

- The EI uses the *libconfig* to parse the input file.

```
// Example
component: {
   package: {// package
      library: {
         model = “3d-ice”;
         ambient_temperature = 300.0;
         grid_rows = 100;
         /* 3D-ICE PARAMETERS */
      };

      component: {
         core0: {// package.core0
                  library: {
                     model = “none”;
                     voltage = 0.8;
                     clock_frequency = 2.0e9;
               /* COMMON PARAMETERS OF core0 */
                  };

                  component: {
                     data_cache: {
                     };
                  };
         }; // package.core0
      };
   }; // package
};
```

```
// Continued with core.data_cache
component: {
   data_cache: {// package.core0.data_cache
      library: {
         model = “mcpat”;
         energy_model = “array”;
         energy_submodel = “cache”;
         line_sz = 64;
         assoc = 4;
         /* MCPAT PARAMETERS */
         /* NO NEED TO REDEFINE voltage */
      };
   }; // package.core.data_cache

   inst_dec: {// package.core0.inst_dec
      library: {
         model = “mcpat”;
         energy_model = “inst_decoder”;
         x86 = true;
         /* MCPAT PARAMETERS */
      };
   }; // package.core0.inst_dec
...
};
```

User API Example (1)

- The simulator creates the Energy Introspector.
- Each manifold model has the pointer to the Energy Introspector.

```
// Example – Serial Simulation
/* MAIN FUNCTION: CREATE AND CONFIGURE THE ENERGY INTROSPECTOR */
energy_introspector_t *ei = new energy_introspector_t("input.config"); // Create EI
ei->configure(); // Configure EI

...

/* CONNECT EI INTERFACE – SERIAL SIMULATION */
Core_t *core = Component :: GetComponent<core_t>(ManifoldCoreID);
core->connect_EI(ei);
core->set_sampling_interval(0.0001);

...
Manifold :: Run();
...
```

User API Example (2)

- The EI provides several API functions to use the models.

```
/* WITHIN A MANIFOLD MODEL */
Comp_ID package_id, core_id, data_cache_id;
...
/* IDs ARE REQUIRED FOR ALL COMPONENTS WHERE DATA ARE MONITORED */

counter_t data_cache_counter; // Data cache counter
...
/* COUNTERS ARE DEFINED FOR ALL COMPONENTS WHERE POWER IS CHARACTERIZED */

void core_t::connect_EI(energy_introspector_t *ei) {
  energy_introspector = ei;
  // Get the pseudo component ID of modeled processor components
  package_id = energy_introspector->get_component_id("package");
  assert(package_id != INVALID_COMP_ID);
  ...
}

void core_t::tick() {
  if(NowTicks() % sampling_interval == 0) {
    /* ARCHITECTURE SIMULATION IS PERFORMED, AND COUNTERS ARE COLLECTED FOR DATA CACHE */
    energy_introspector->calculate_power(data_cache_id,time,period,data_cache_counter);
    /* calculate_power() IS CALLED FOR ALL COMPONENTS WHERE POWER IS CHARACTERIZED */
    energy_introspector->calculate_temperature(time,period,/*PowerSync?*/true);

    int err; Kelvin core_temp; // Data are already sync'ed
    err = energy_introspector->pull_data(core_id,time,period,EI_DATA_TEMPERATURE,&core_temp);
  }
}
```

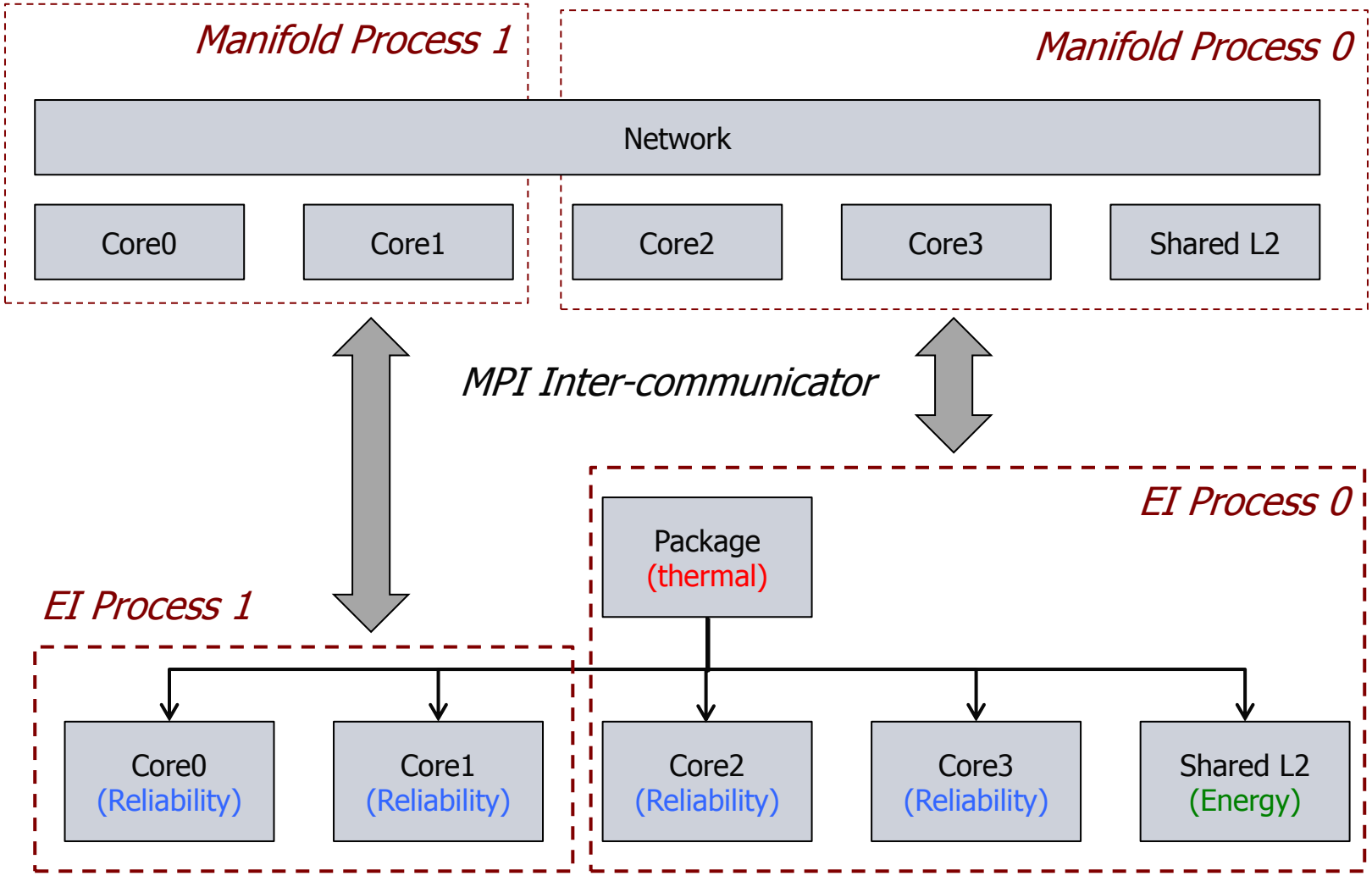
DEMONSTRATION

<*Architecture-physics co-simulation via Manifold and Energy Introspector*>

```
// Trace file from the simulation

time= 124.0 | core0
        instantIPC= 0.997120 , avgIPC= 1.008513
        power= 2.195946 W (dynamic= 1.299701 W, leakage= 0.896246 W, area= 5.28mm^2)
        clk= 2.000 GHz, temp= 332.3, failure_rate= 1.934528e-11 (MTTF= 1.269645)

time= 124.0 | core1
        instantIPC= 1.088155 , avgIPC= 1.089841
        power= 2.325650 W (dynamic= 1.418857 W, leakage= 0.906793 W, area= 5.28mm^2)
        clk= 2.000 GHz, temp= 335.9, failure_rate= 2.228109e-11 (MTTF= 1.102353)

time= 124.0 | core2
        instantIPC= 1.372630 , avgIPC= 1.239406
        power= 2.698654 W (dynamic= 1.791860 W, leakage= 0.906793 W, area= 5.28mm^2)
        clk= 2.000 GHz, temp= 335.8, failure_rate= 2.189546e-11 (MTTF= 1.121768)

time= 124.0 | core3
        instantIPC= 1.124560 , avgIPC= 1.050577
        power= 2.361293 W (dynamic= 1.465048 W, leakage= 0.896246 W, area= 5.28mm^2)
        clk= 2.000 GHz, temp= 333.0, failure_rate= 1.958338e-11 (MTTF= 1.254208)
```

PARALLEL INTERFACE:

<*Parallel Simulation via MPI interface*>



Parallel EI Configuration

- The EI supports *parallel simulation* via the *MPI interface*.
- The *pseudo component hierarchy* can be partitioned into *multiple MPI processes*.

```
// Example – Rank0
component: {
   package: {// package
      library: {
         model = “3d-ice”;
         ambient_temperature = 300.0;
         grid_rows = 100;
         /* 3D-ICE PARAMETERS */
      };
      component: {
         core0: {// package.core0
                  remote = true;
         }; // package.core0
         core1: {// package.core1
                  remote = true;
         }; // package.core1
         core2: {// package.core2
                  model = “none”;
         }; // package.core2
...
      };
   }; // package
};
```

```
// Example – Rank1
component: {
   package: {// package
      library: {
         remote = true;
      };
      component: {
         core0: {// package.core
                  library: {
                     model = “none”;
                     voltage = 0.8;
                     clock_frequency = 2.0e9;
               };
               ...
         }; // package.core
      };
   }; // package
};
```

Parallel API Example (1)

- The MPI process creates the Energy Introspector.

```
// Example – Parallel Simulation
/* MAIN FUNCTION: CREATE AND SPLIT THE MPI COMMUNICATOR FOR MANIFOLD AND EI */
MPI_Comm INTER_COMM, LOCAL_COMM;
MPI_Comm_split(MPI_COMM_WORLD,(MyRank<Manifold_NPs),Rank,&LOCAL_COMM);
MPI_Intercomm_create(LOCAL_COMM,0,MPI_COMM_WORLD,MyRank<Manifold_NPs?
                Manifold_NPs:0,1234,&INTER_COMM);

/* CREATE AND CONFIGURE THE ENERGY INTROSPECTOR */
energy_introspector_t *ei = new energy_introspector_t("input-rank0.config",
                                        INTER_COMM,LOCAL_COMM);
ei->configure(); // Configure EI

...
Manifold :: Init(argc,argv,&LOCAL_COMM); // Manifold runs within its MPI communicator
...

/* INSTANTIATE EI CLIENT – PARALLEL SIMULATION */
Core_t *core = Component :: GetComponent<core_t>(ManifoldCoreID);
core->connect_EI(&INTER_COMM, Rank);
core->set_sampling_interval(0.0001);

...
Manifold :: Run();
...
```

Parallel API Example (2)

- The EI client is used to communicate over MPI processes.

```
/* WITHIN A MANIFOLD MODEL */
void core_t::connect_EI(MPI_Comm *InterComm, int EIRank);
  // Create EI client interface
  EI_client_t *EI_client = new EI_client(InterComm,EIRank);

  // Get the pseudo component ID of modeled processor components
  package_id = EI_client->get_component_id("package");
  assert(package_id != INVALID_COMP_ID);
  ...
}

void core_t::tick() {
  if(NowTicks() % sampling_interval == 0) {
    /* ARCHITECTURE SIMULATION IS PERFORMED, AND COUNTERS ARE COLLECTED FOR DATA CACHE */
    EI_client->calculate_power(data_cache_id,time,period,data_cache_counter);
    /* calculate_power() IS CALLED FOR ALL COMPONENTS WHERE POWER IS CHARACTERIZED */
    EI_client->calculate_temperature(time,period,/*PowerSync?*/true);

    Kelvin core_temp; power_t core_power; // Data are already sync'ed
    int err = EI_client->pull_data(core_id,time,period,EI_DATA_POWER,&core_power);
    err = EI_client->pull_data(core_id,time,period,EI_DATA_TEMPERATURE,&core_temp);
  }
}
```

EI: Summary

- The Energy Introspector is an *enabler* for architecture-physics exploration.
- Any *new models* can be incorporated into the interface, and we plan to add more models.
- For an access to the latest EI:

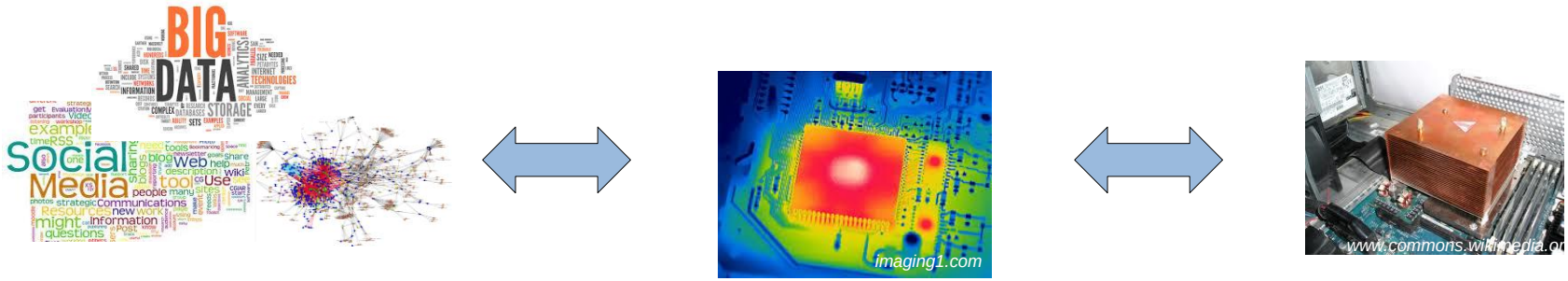
svn co
https://svn.ece.gatech.edu/repos/Manifold/trunk/code/models/energy_introspector/

Outline

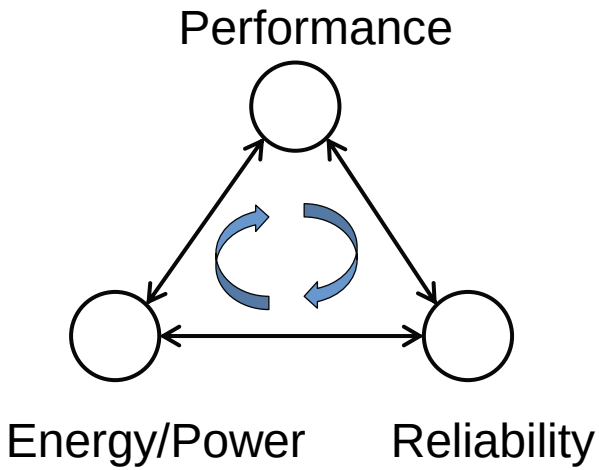
- Introduction
- Execution Model and System Architecture
- Multicore Emulator Front-End
- Component Models
  - Cores
  - Network
  - Memory System
- Building and Running Manifold Simulations
- Physical Modeling: Energy Introspector
- Some Example Simulators

What can Manifold Enable?

- Manifold enables cross-disciplinary evaluations
- Applications ←→ Power ←→ Thermal ←→ Cooling



- Multi-scale simulation → cycle-level to functional
- Tradeoff studies



Some Example Simulators

- Power capping studies
- Reliability studies
- Workload ←→ Cooling interaction

Power Capping: Simulation Model

SIMULATED PROCESSOR CONFIGURATION

Parameters	Out-of-order Core	In-order Core
Architectural Configuration		
ISA	x86 IA32	
Pipeline Depth	20 stages	16 stages
Fetch/Decode	4 instructions	2 instructions
Execution	6 ports	3 ports
L1 Cache	4-way 32KB	4-way 32KB
L2 Cache	8-way 512KB	8-way 512KB
Physical Configuration		
Clock Frequency	1.85-3.75GHz	
Supply Voltage	0.6-1.0V	
Feature Size	45nm	

- Controller gain is adjusted every 5 ms
- Each core has its own core and power budget – two OOO and two IO cores.

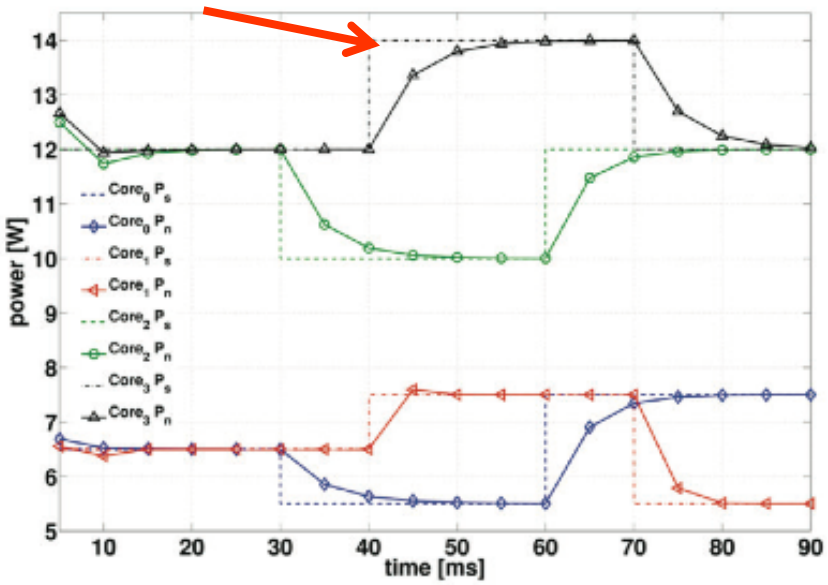
POWER TRACKING PHASE FOR ASYMMETRIC PROCESSOR

Core	Phase 1	Phase 2	Phase 3
$Core_0$ (in-order)	6.5 W	5.5W	7.5W
$Core_1$ (in-order)	6.5 W	7.5W	5.5W
$Core_2$ (out-of-order)	12 W	10W	12W
$Core_3$ (out-of-order)	12 W	14W	12W

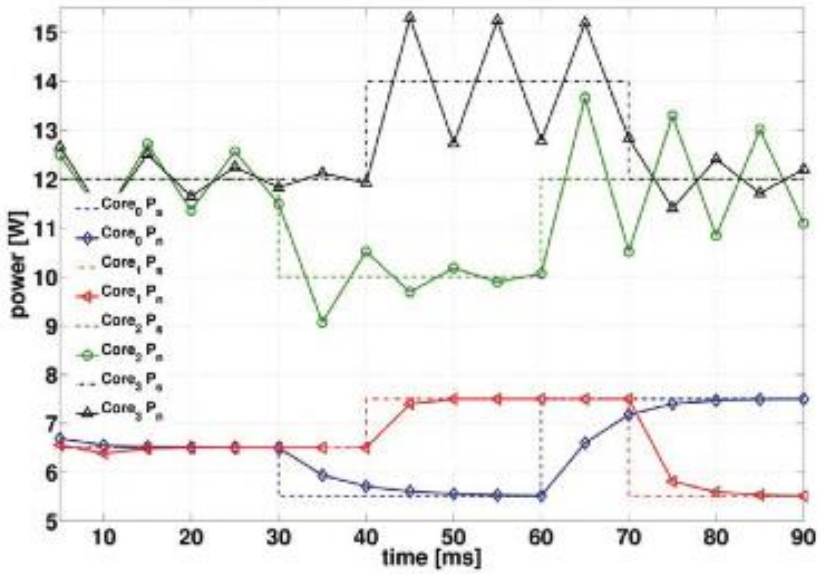
Power Targets

Power Capping Controller

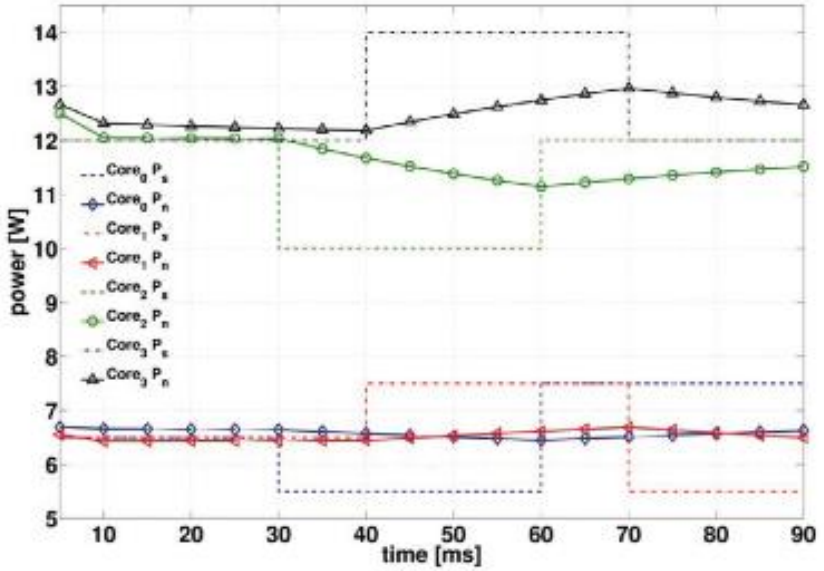
New set point



(a) Adaptive gain controller



(b) High fixed gain controller (K = $500e^6$)

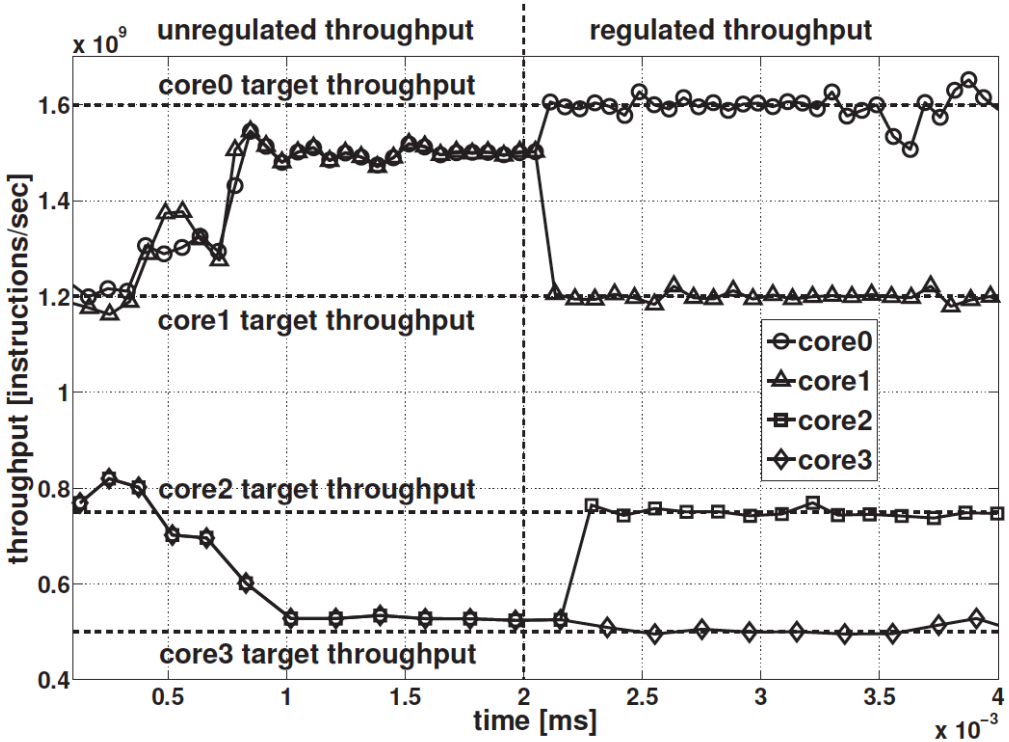


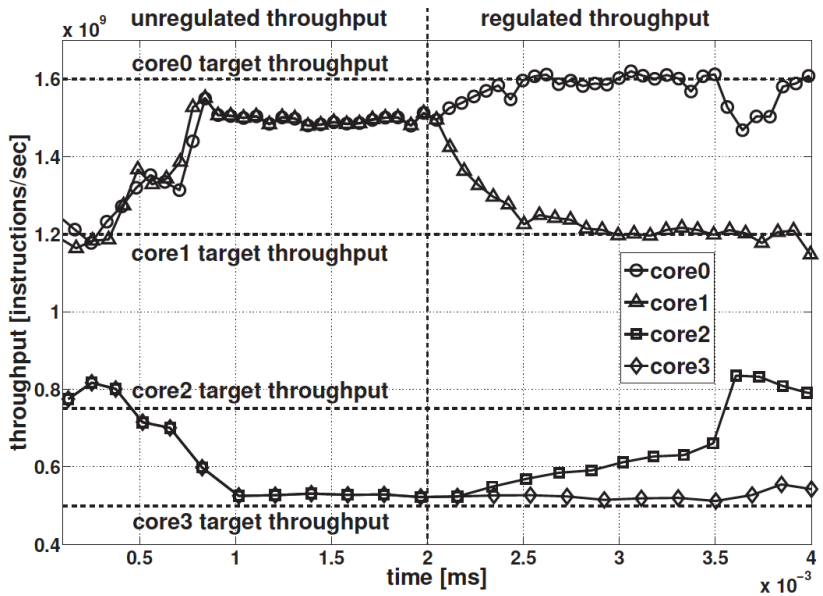
(c) Low fixed gain controller (K = $25e^6$)

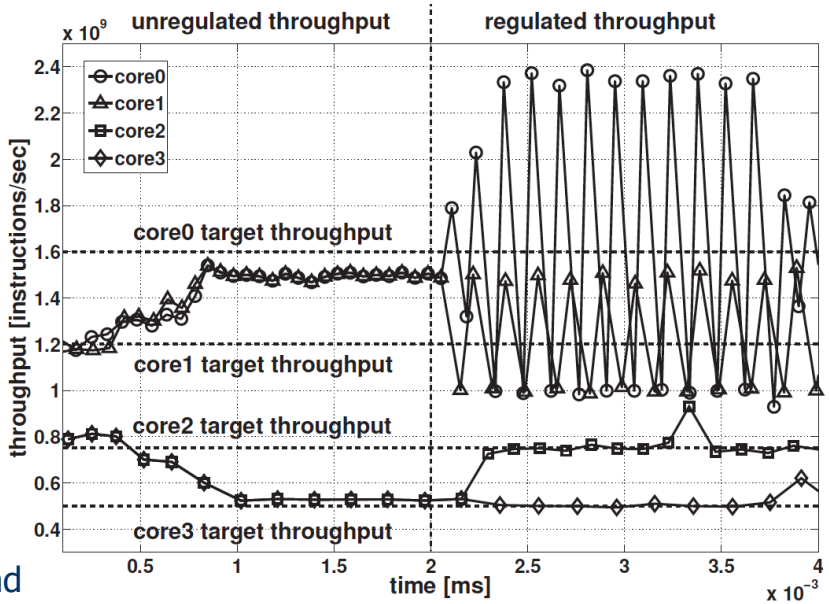
- High fixed-gain controller over-reacts to high power cores, whereas low fixed-gain control is slow to react to low power cores.

N. Almoosa, W. Song, Y. Wardi, and S. Yalamanchili, "A Power Capping Controller for Multicore Processors," American Control Conf., June 2012.

Throughput Regulation: Adaptive



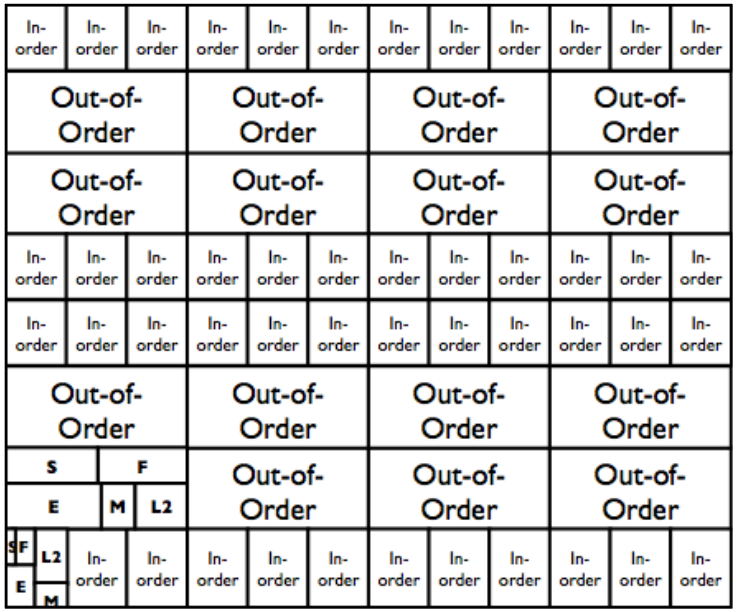




- High fixed-gain controller over-reacts to high power cores, whereas low fixed-gain control is slow to react to low power cores.

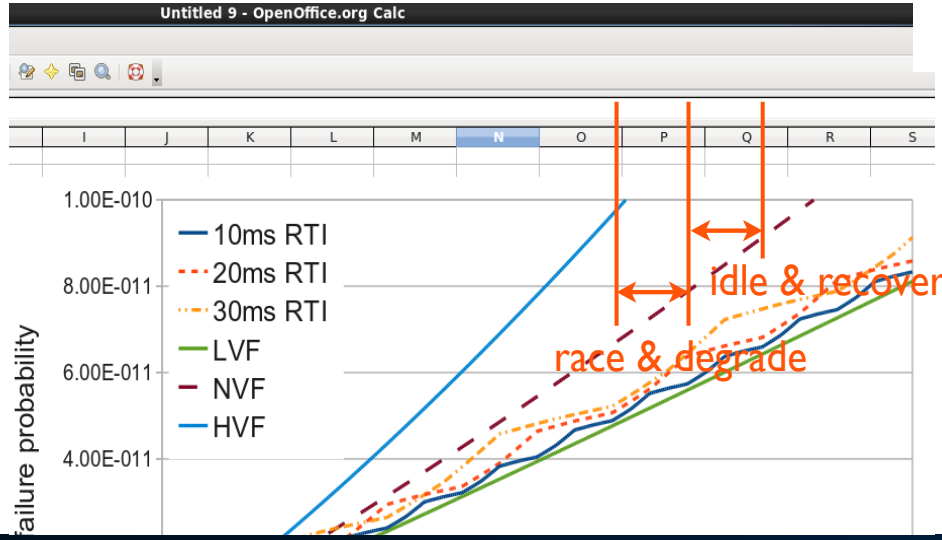
N. Almoosa, W. Song, Y. Wardi, and S. Yalamanchili, "Throughput Regulation on Multicore Processors via IPA," 2012 IEEE 51st Annual Conference on Decision and Control (CDC)

Adaptation to Aging and Reliability

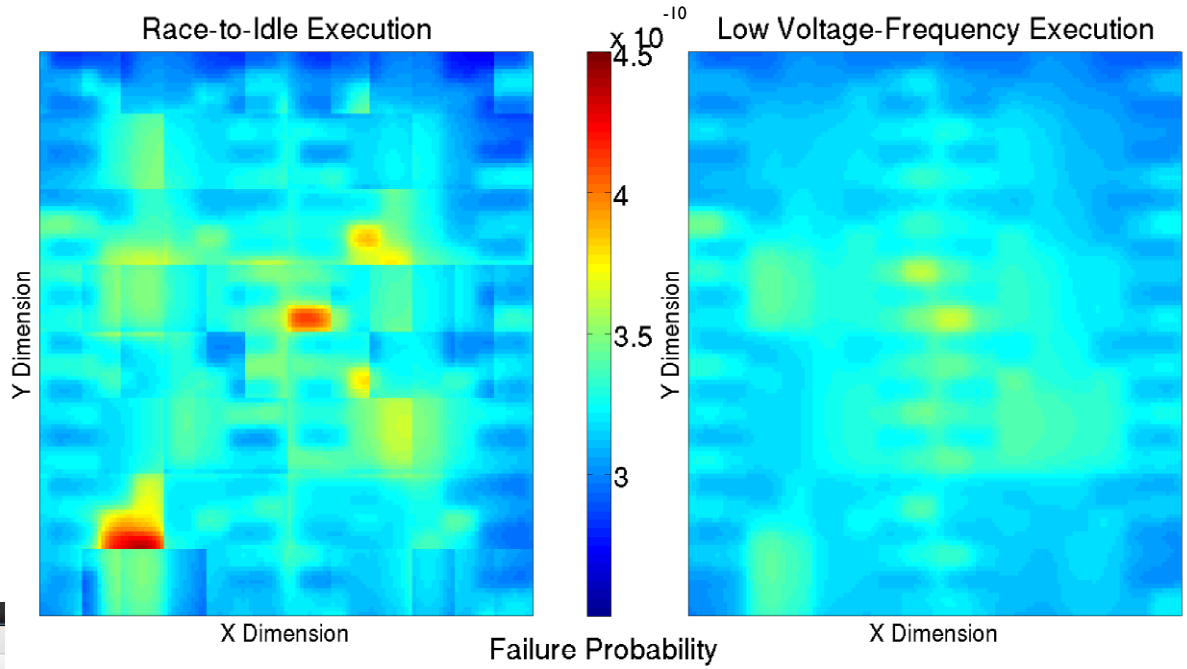


NVF: Nominal Voltage Frequency

LVF: Low Voltage Frequency HVF: High Voltage Frequency



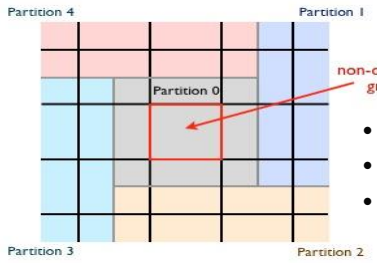
64-core asymmetric processor floor plan



Failure probability comparison between per-core race-to-idle executions (left) and continuous low-voltage executions (right)

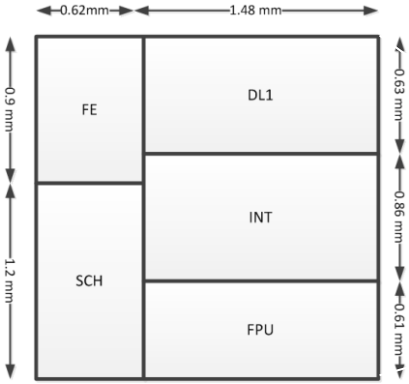
Transient race-to-idle executions vs. continuous executions

Workload-Cooling Interaction

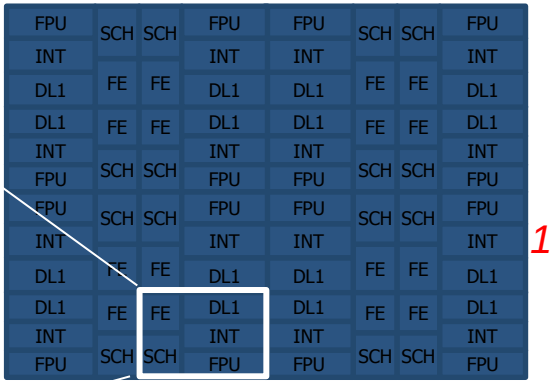


- Thermal Grids: 50x50
- Sampling Period: 1us
- Steady-State Analysis

2.1mm x 2.1mm



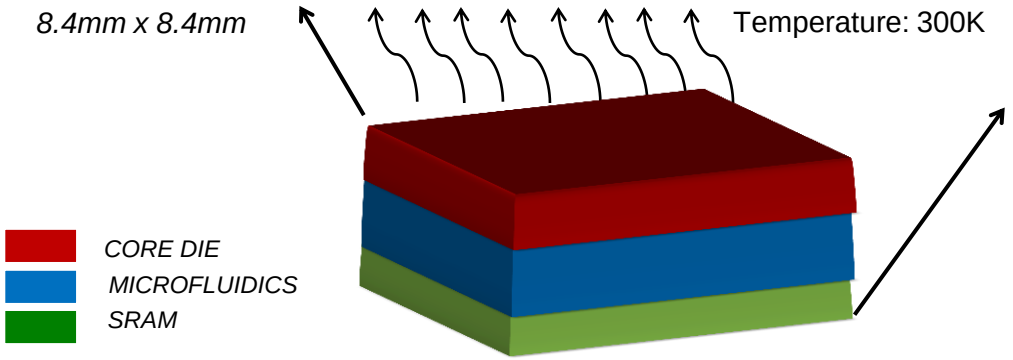
Nehalem-like, OoO cores;
3GHz, 1.0V, max temp 100°C
DL1: 128KB, 4096 sets, 64B
IL1: 32KB, 256 sets, 32B, 4 cycles;



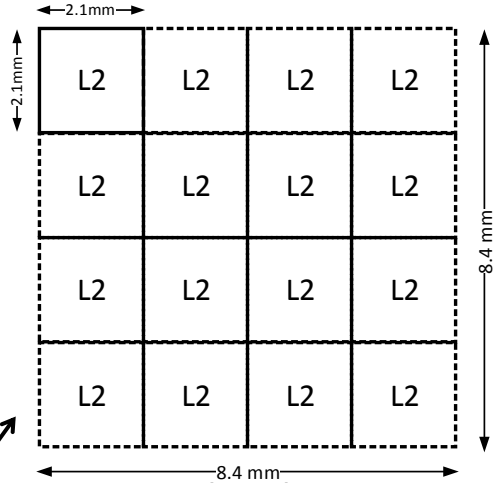
8.4mm x 8.4mm

16 symmetric cores

Ambient:
Temperature: 300K



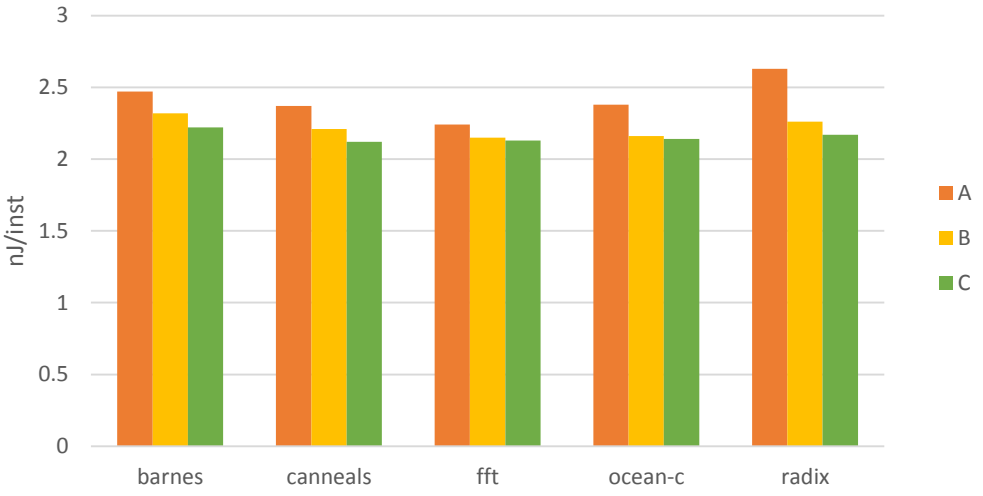
CORE DIE
MICROFLUIDICS
SRAM

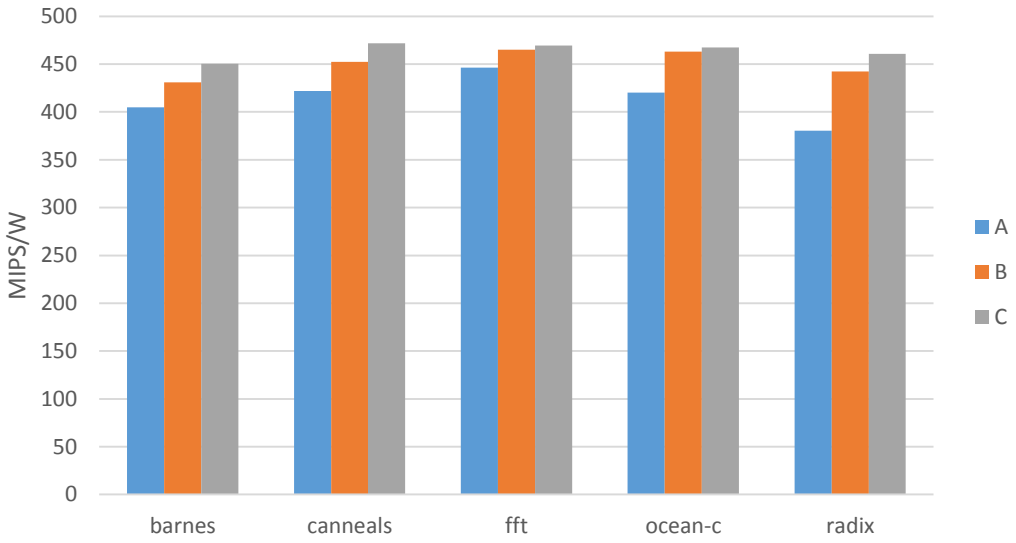


L2 & Network Cache Layer:
L2 (per core): 2MB, 4096 sets, 128B, 35 cycles;
DRAM: 1GB, 50ns access time (for performance model)

Coolant/Configuration	A	B	C
Flow rate (ml/min)	7	42	84
Top Heat Coeff (W/um^2-K)	2.05e-8	5.71e-8	8.01e-8
Bot. Heat Coeff (W/um^2-K)	1.69e-8	4.72e-8	6.63e-8

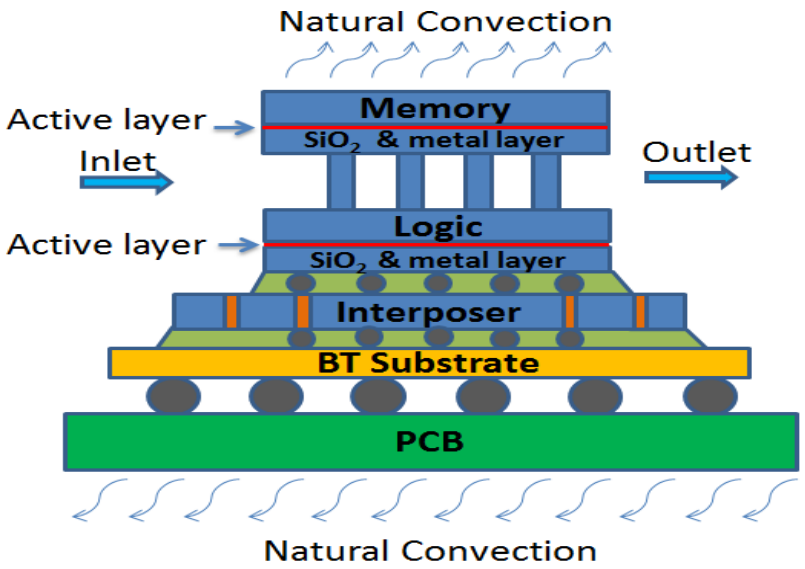
Impact of Flow Rate & Workload on Energy Efficiency



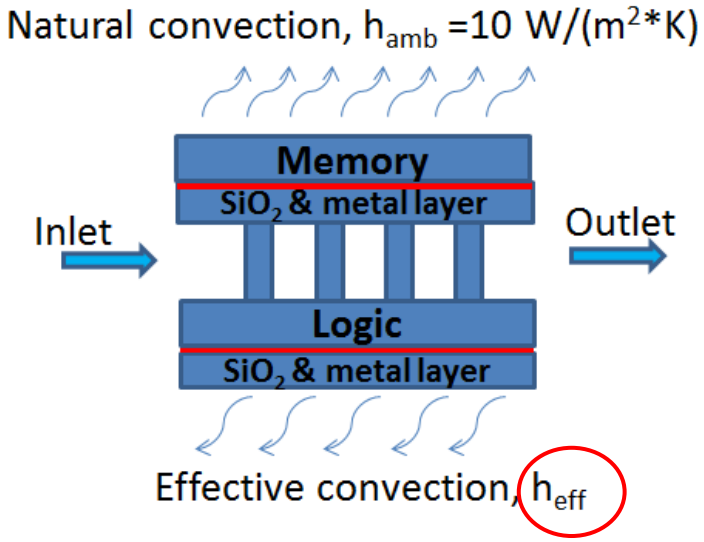


- Memory bound applications benefit more than computation bound applications
- Overall energy improvement
  - 4.9%-17.1% over 12X increase in flow rate
  - 4.0%-14.1% over 6X increase in flow rate
- Does not include pumping power

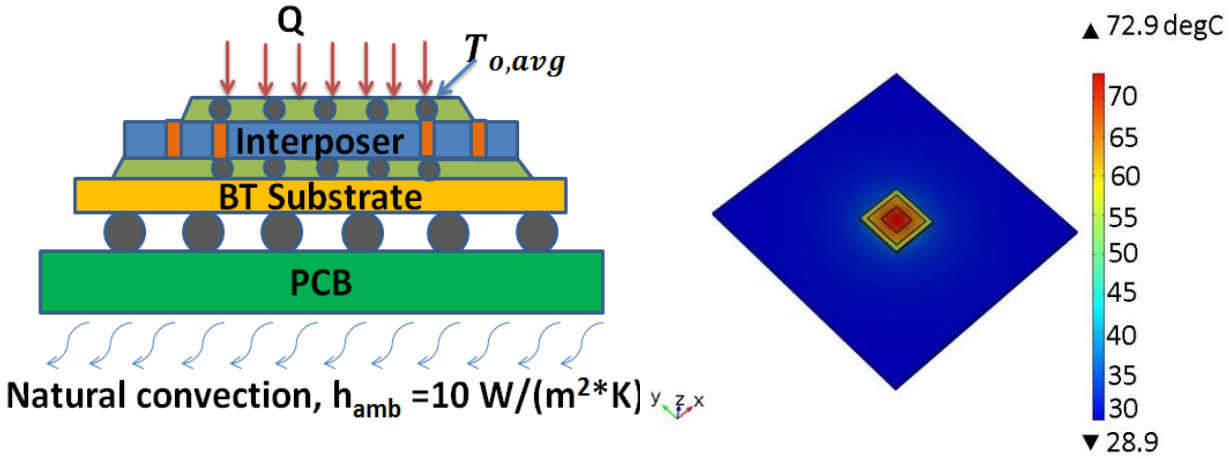
3D Stacked ICs Structure Model



3D stacked ICs structure



Simplified structure



Conduction FE model and temperature results

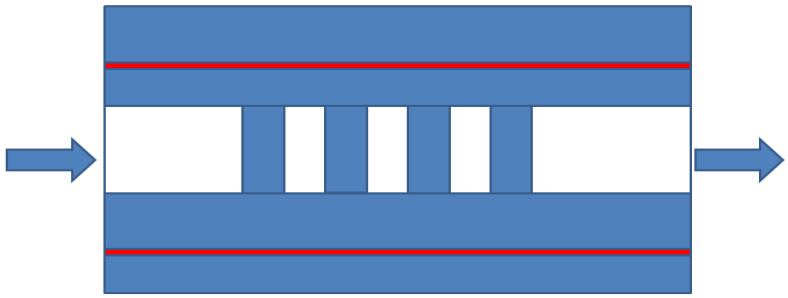
Effective heat transfer coefficient is obtained by FE model on the left:

h_{eff}=562.4 W/m^2*K

Z. Wan et. al., IEEE Therminic 2013, Berlin, 25. -27. Septemeber 2013 (accepted)

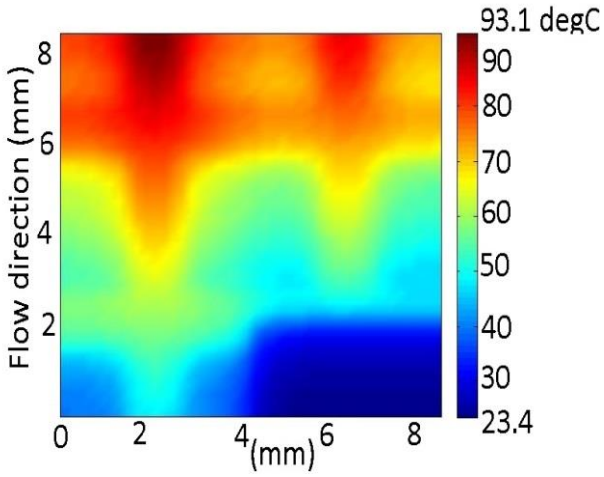
Case Study with Different Microgap Configurations

Microgap configurations

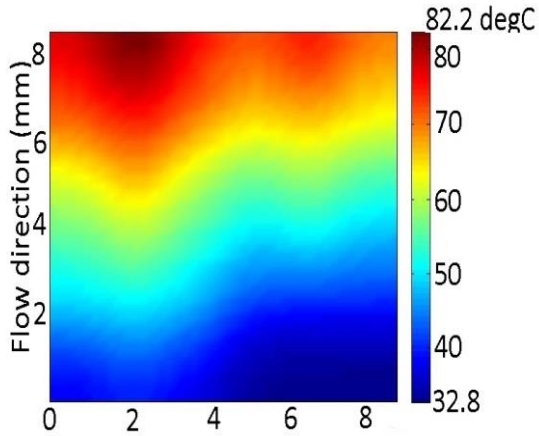


Configuration 1: One microgap

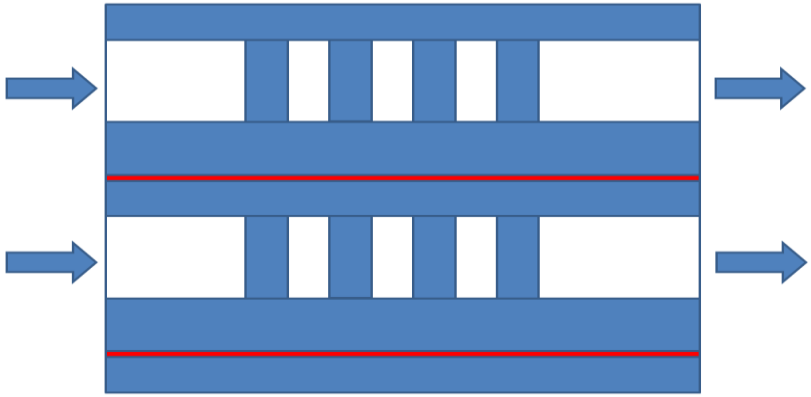
Logic tier



Memory tier



Temperature results: One microgap, logic tier at bottom and memory tier on the top



Configuration 2: Two microgaps

Results for different cases

Pump power: 0.03 W	Configuration			$T_{max,logic}$ (℃)	$T_{max,memory}$ (℃)
	Micro-gap	Top	Bottom		
Case 1	1	M	L	93.1	82.2
Case 2	1	L	M	114.9	77.1
Case 3	2	M	L	87.7	54.8
Case 4	2	L	M	72.7	58.3

Summary

www.manifold.gatech.edu

Not to provide a simulator,
but
Composable simulation infrastructure for constructing multicore simulators, and

Provide base library of components to build useful simulators