SESC: SuperESCalar Simulator

Pablo Montesinos Ortego
Paul Sack

December 20, 2004

1 What is this all about?

The biggest challenge for new students in architecture research groups is not passing theory or software classes. It is not finding a new apartment or registering with the INS. It is understanding the architecture of the processor simulator that will soon confront them–a simulator coded not for perfection, but for deadlines. Even the most well-conceived simulator can quickly look like a Big Ball of Mud to the unitiated.

1.1 Typical initation procedure

- Find a desk
- Configure computer, email userhelp, etc.
- Advisors says something like "Why don't you start working on this?"
- Bother older student for simulator access
- Download source code
- Find documentation
- After finding that the documentation is less than useless (if it even exists), bother senior students with pesky questions
- Senior students graduate
- You are the most senior student
- Give up, read the code
- Add more mud
- ...
- Graduate

1.2 Purpose of this document

This document is intended to break the chain of simulator initiation. This document explains, at a high level, the workings of the core of the simulator, the part that new students must learn first. SESC is under constant development and changes overnight, but this document describes the more permanent, stable core of the simulator.

This document cannot explain all of SESC. It is a great starting point, from which the SESC novice can dive into the source code.

This is the documentation we wish we had.

2 Introduction

2.1 What is a microprocessor simulator?

In microarchitecture research, typically researchers have some kind of proposal for a microprocessor that will be better than the current state of the art. It might be faster, use less power, be more reliable, or create the perfect loaf of bread. In any event, since it is expensive to design and fabricate a microprocessor, researchers write microprocessor simulators. There are different approaches to this. Some are trace-driven, i.e., they use instruction traces of applications. Most are execution-driven, i.e., they actually execute the simulated application.

Many simulators also divide simulation into an emulator, which actually executes the simulated application, and a timing simulator, which models the timing and energy of the simulated application.

It is also common to have at least part of the simulator be event-driven. What this means is that parts of the simulator can schedule an event, i.e., a function call with parameters, to occur at some time in the future.

2.2 What is SESC?

SESC is a microprocessor architectural simulator developed primarily by the i-acoma research group at UIUC and various groups at other universities that models different processor architectures, such as single processors, chip multiprocessors and processors-in-memory. It models a full out-of-order pipeline with branch prediction, caches, buses, and every other component of a modern processor necessary for accurate simulation.

SESC is an event-driven simulator. It has an emulator built from MINT, an old project that emulates a MIPS processor. Many functions in the core of the simulator are called every processor cycle. But many others are called only as needed, using events.

2.3 The Architecture Business Cycle of SESC

This section describes SESC's Architectural Business Cycle as shown below:

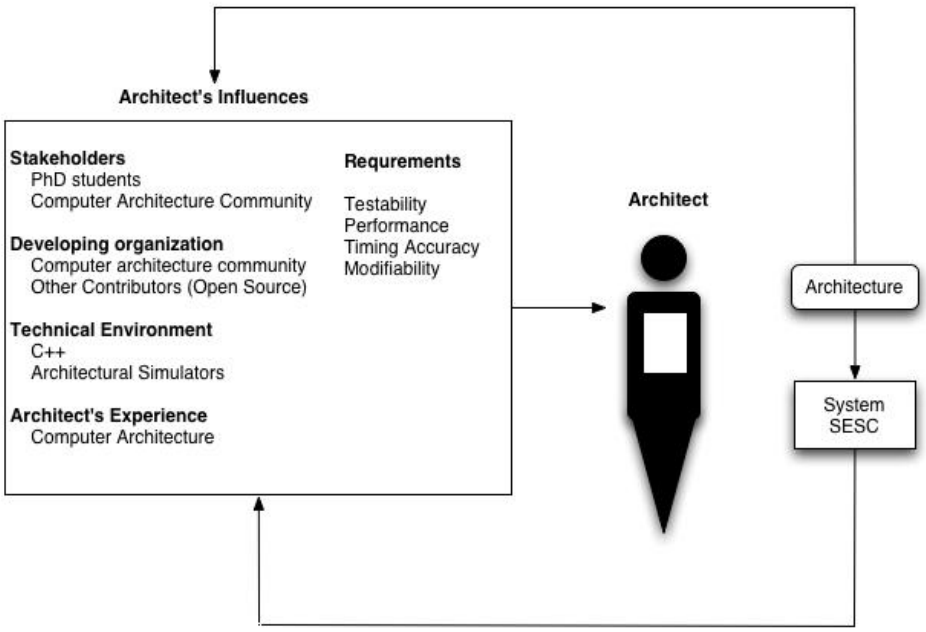

Figure 1: The Architectural Business Cycles of SESC.

2.4 Stakeholders

SESC's stakeholders are PhD students and researchers in computer architecture. Its configurability and the availability of the source code make it an excellent tool for evaluating research proposals. It has several advantages over other simulators, such as Simplescalar. For one, SESC models a variety of architectures, including dynamic superscalar processors, CMPs, processor-in-memory, and speculative multithreading architectures. Simplescalar focuses on single processors only. In addition, SESC is very fast, capable of executing over 1.5 millions instructions per second (MIPS) on an Intel Pentium 4 3 GHz processor.

Apart from our group, it is used and developed by other research groups at the University of Illinois, University of California at Santa Cruz, University of Rochester, North Carolina State University, Georgia Institute of Technology, and Cornell University.

2.5 Developing Organization

SESC's primary developers are Jose Renau at the University of California at Santa Cruz and some of the students in the i-acoma group at the UIUC. Because the software is under an open-source license, anyone can participate in its development. In fact, users are encouraged to submit their changes so that SESC improves. Changes to the core of the simulator, that are useful for all

the users are usually commited once the code has been verified to work. Other changes that specifically support research ideas are usually not commited until the research has been published. This is done so that each research group does not have to disclose its projects.

Not all submitted code will be accepted, though. In order to be accepted, any upgrade is reviewed by Jose Renau, and will be rejected it if it does not achieve minimum quality and code-style standards.

2.6 Technical Environment

SESC is written in C++, since it is faster than Java and has good object-oriented programming support. SESC runs on many UNIX sytems as Linux and Darwin/MacOS X. SESC runs on big-endian and little-endian processors.

2.7 Architect's Experience

C++ was chosen not only for performance reasons, but also because the primary architect was very familiar with it. All the main contributors have a strong background in using or writing architecture simulators.

2.8 Requirements and Qualities

Modern microprocessor simulators are highly complex software systems. They have rigorous timing requirements and must maintain a high fidelity of simulation accuracy. Fidelity should be the key requirement of any simulator. After all, there is no point on having a simulator that does not reflect the microprocessor behavior accurately.

Second to accuracy is performance. Since a simulator executes instructions on the order of one million times slower than a real processor, execution speed is important. A faster simulator will be able to execute benchmarks faster. A single research paper will often require thousands of hours of simulation time, and reducing this is important, as reserach group's have limited computing budgets.

Figure 2 above shows the architect's influences on SESC. Its end-users are PhD students that are trying to test new ideas without building the actual hardware. The figure shows how an architecture research group works. At some point in the semester, an important conference hsa a call for papers. Then, different students propose ideas for paper submissions. Once they are approved by the group advisor, each student or group of student will create his own version of the simulator. During the next several months, he will modify and extend the simulator in order to support the paper. Once the paper is submitted, there is a period of time when the students integrate their branches to create a new version of the simulator. Integration is not an easy task. In many cases, two students work on similar ideas and modify the same modules.

Therefore, Figure 2 suggests two requirements that SESC should achieve. First, it has to be modifiable. If every student on the group is going to test his

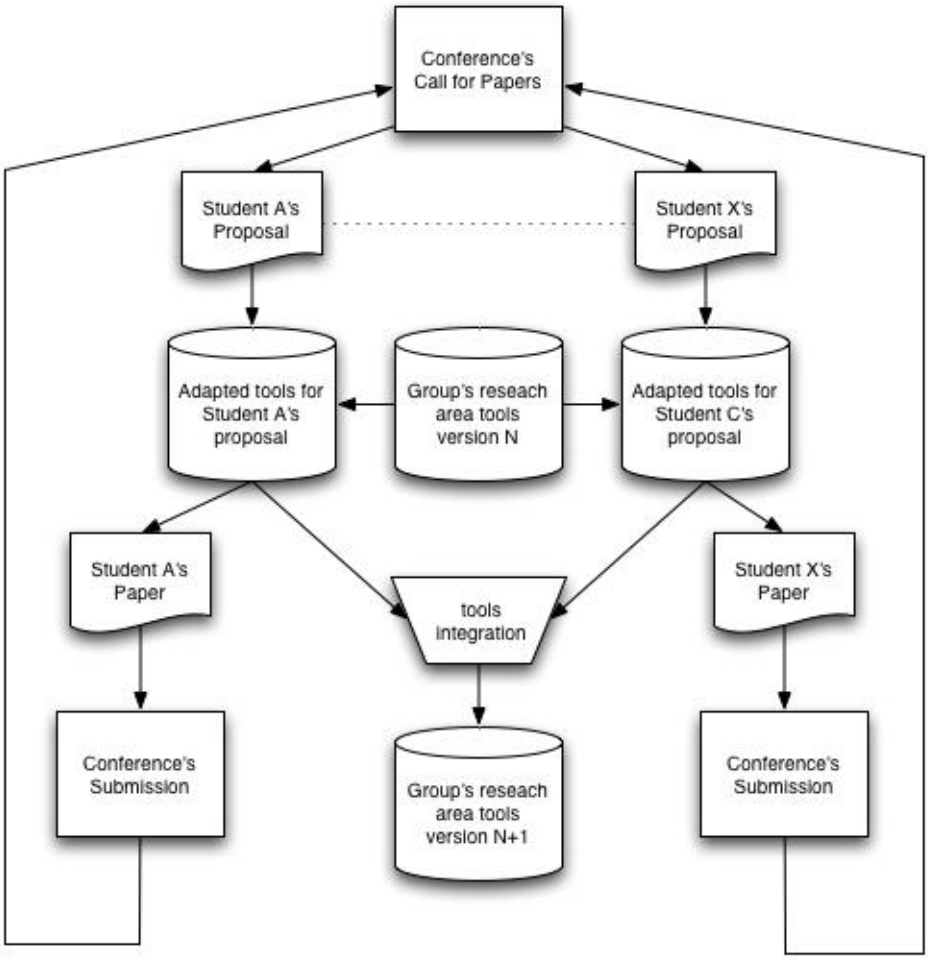

Figure 2: The ABC of the i-acoma group.

ideas using SESC, its code has to be highly modifiable. High modifiability might not be an important requirement in pre-production environments: changes is real products are incremental, as are the changes in the simulator. Second, the code has to be easy to test. Integration phases can take weeks if there is too much overlap amongst the different branches. In fact, it is likely that not all the changes in a branch will ever become part of the main source code tree.

Many of the available microprocessor simulators offer tons of features but have a drawback: they are very slow. Speed is always important, but especially during the two weeks before the conference's deadline. SESC was designed with performance in mind: it is up to ten times faster than competing simulators.

2.9 What is a superscalar out-of-order pipeline?

We can envision a simple processor as a black box that executes one instruction every cycle. Early microprocessors followed this approach. However, researchers understood that no instruction used all the elements inside the processor all the time, and that it could be possible for a processor to simultaneously operate on more than one instruction at a time: this is the concept behind a pipeline. Pipelining is a microprocessor technique that allows multiple instructions to be overlapped while they are being executed inside the processor. Pipelining a processor is not an easy task. There are many challenges that pipelining introduces, e.g., exception handling and branch misprediction. However, all major processors these days are heavily pipelined. The latest Intel Pentiums have over 30 pipeline stages.

It is possible to build a microprocessor in which each stage of the pipeline requires one cycle. In that situation, a microprocessor can be seen as a FIFO queue: the first instruction fetched is the first instruction committed. Still, common sense reasons that performing a floating point multiplication will take more time than comparing an integer with zero. However, supporting multi-cycle operations results in a pipeline that is no longer a FIFO, as an instruction fetched at time t can finish before than an instruction fetched at time *t-1*. Again, out-of-oder processors have to deal with many problems when handling exceptions.

Under perfect conditions, an out-of-order pipelined microprocessor would retire one instruction per cycle. To improve performance further we could allow multiple instructions to be retired in a clock cycle. This can be done in two ways:

- Very long instruction word architectures (VLIW): the processor issues a fixed number of instructions per cycle. The compiler schedules them.
- Superscalar processors: the processor tries to issue more than one instruction per cycle so as to keep all of the functional units busy. There may be limitations on parallel issue, like no more than one memory instruction per clock cycle. In order to maximize the number of instructions issued per clock, both static and dynamic scheduling techniques are used. Statically

scheduled processors use in-order execution, while dynamically scheduled processors use out-of-order execution.

2.10 How does SESC model this?

In SESC, the actual instructions are executed in an emulation module, which emulates the MIPS Instruction Set Architecture (ISA). It emulates the instructions in the application binary in order. The emulation module is built from MINT, a MIPS emulator.

The emulator returns instruction objects to SESC which are then used for the timing simulator. These instruction objects contain all the relevant information necessary for accurate timing. This includes the address of the instruction, the addresses of any loads or stores to memory, the source and destination registers, and the functional units used by the instruction. The bulk of the simulator uses this information to calculate how much time it takes for the instruction to execute through the pipeline.

The justification for this is twofold. First, it is much faster to have the actual instruction executed in a simple emulator. Second, it is easier to program and debug when execution and timing are separated. The timing simulator, which is very complex, does not need to be 100% accurate if it does not affect the computation of instructions. If a bug causes the simulator to have a 0.1% error in timing accuracy, this is perfectly acceptable. In some instances, the programmers of SESC could deliberately ignore extremely rare race conditions which would be difficult to program correctly and would have minimal impact on timing.

3 Detailed view

In this section, we describe how an instruction in an application binary flows through the simulator.

3.1 Instruction types

There are two instruction types in SESC. The first represents actual instructions in the application binary. The second represents short-lived instructions as they flow through the pipeline.

3.1.1 Static Instructions

These are also referred to as **static instructions**. An example would be `add r1,r2,3`. This is implemented in the Instruction class. Instructions are created during SESC initialization. A function reads the application binary and decodes each instruction into an internal representation. The internal representation contains information to make executing the instruction fast.

The instruction object contains the source registers, the instruction type (branch, load, store, integer arithmetic, or floating point arithmetic), a pointer

to a function to execute the specific instruction (such as floating-point addition), a pointer to the next instruction to execute, and a pointer to a branch target if one exists.

It is crucial that emulation is fast, as often benchmarks will have a lengthy initialization period that can be run without a timing model.

3.1.2 Dynamic Instructions

Dynamic instructions are specific instances of a static instruction. Each time a static instruction is executed, a dynamic instruction is created. For example, a dynamic instruction might refer to the above-mentioned add instruction when `r1` contains the value 10 and `r2` contains the value -8 and the processor is on clock cycle 32948.

Dynamic instructions are implemented in the DInst class. DInst objects have many fields.

Consider the following sequence of instructions:

```
1: ld r3
2: ld r4
3: add r3,r4,r5
```

Instruction 3 is said to be dependent upon instructions 1 and 2, since it gets its input operands from those instructions. The DInst class keeps track of this by linking the DInst for instruction 3 to DInsts 1 and 2 and linking DInsts 1 and 2 to Dinst 3. These relationships are referred to as **dependencies**.

There are many other variables which keep track of whether a DInst has been issued, executed, or retired; which CPU the DInst belongs to; the sequence number of the DInst; which resources a DInst needs for execution (such as a floating-point multiplier); and whether a DInst is executing in the wrong path of a mispredicted branch.

This class is extended for many specific projects in ways which will not be documented here.

3.2 Emulation

Emulation is controlled by the ExecutionFlow class. The upper-level interface to ExecutionFlow is through the executePC() function. The executePC() function executes the next instruction and returns the corresponding DInst.

The executePC() function actually calls a function, exeInst(). This function performs some checks, such as that the address of the instruction is a legal address, and then executes the instruction. Each Instruction object contains a pointer to a function that emulates the instruction. The source and destination operands of the instruction are also kept in the Instruction object. Thus, the actual execution is quite quick.

Within the ExecutionFlow class, there is also a function that executes instructions in *rabbit mode* and does not model their timing.

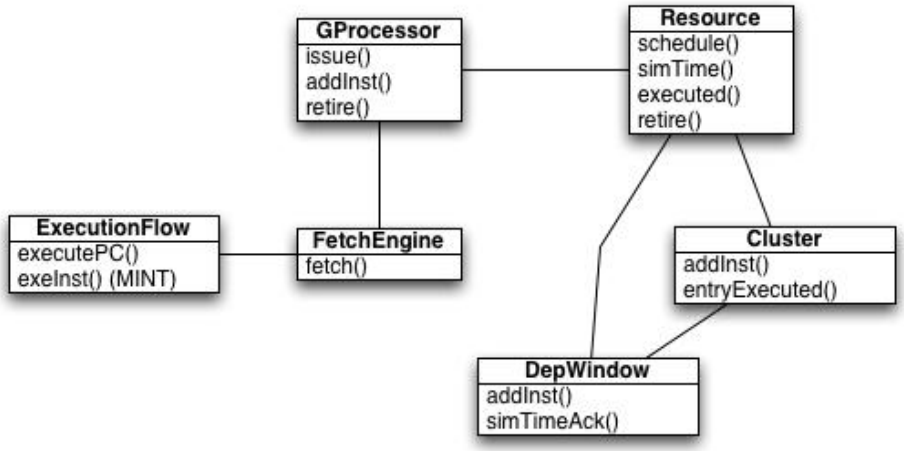

Figure 3: The class interactions that model the pipeline.

Justification

It is very important that emulation without modeling timing be very fast. In many benchmarks there are lengthy initialization sections, which may be longer than the main portion of the program.

In "rabbit mode," in which Instructions are only emulated and no timing simulation is performed, the simulator executes instructions about 1000 times faster than in full simulation mode.

3.3 Pipeline view

In SESC, the GProcessor (generic processor) object type coordinates interactions between the different pipeline stages. The upper-level interface to the GProcessor object is the advanceClock() function. The advanceClock() function advances each stage in the pipeline one clock cycle. It does this by first calling a function to fetch instructions into the instruction queue. It then calls a function to issue instructions from the queue into a scheduling window. There are two *clusters* that schedule and execute instructions, one for integer and one for floating-point instructions, and each has its own scheduling window. Instruction scheduling and execution is handled in other parts of the simulator. Finally, a function is called to retire already-executed instructions from the reorder buffer.

All of the class interactions that model the pipeline are shown in Figure 3.

3.3.1 Fetch/Decode

Instruction fetch is the first stage of the pipeline. In this stage, instructions are brought into the pipeline from the instruction cache. In a typical configuration, the fetch unit will fetch up to 4 instructions per cycle. The fetch unit will also predict which direction a branch will go, and fetch instructions down the predicted path of the branch.

The class that handles instruction fetch is the FetchEngine class. The upper-level interface to this is the fetch() function. The fetch function tries to fetch a configurable number of instructions from the instruction cache. The fetch function also models a branch predictor. The fetch function interacts with the ExecutionFlow class. Specifically, it calls the ExecutePC() function to get the address of the next instruction to fetch from the cache. It then makes a request for this address to the instruction cache. When all of the instructions return from the instruction cache, the fetch *bundle* is passed to the next stage of the pipeline.

Decoding, i.e., transforming instructions from the ISA format into an internal format, is done when the simulated program is read and each instruction in the binary is decoded into an Instruction object. Thus, this class simply adds a decode delay penalty to the time that the bundle is passed to the next stage of the pipeline.

Branch Prediction

SESC supports several different branch predictors. The choice of predictor and its size is selected at run-time. Since branch prediction is done in the fetch unit, the FetchEngine class handles this as well.

If the instruction is a branch, it calls the processBranch() function, which does the branch prediction. If the branch prediction is incorrect, the processBranch() function models the pipeline flush by marking the mispredicted branch.

Successive calls to fetch will call the fakeFetch() function, which fetches instructions from the wrong path of the mispredicted branch. These instructions will be marked as fake instructions.

Finally, when the mispredicted branch executes, it updates the FetchEngine object and restores the correct path of execution.

3.3.2 Issuing & Scheduling

During the issue stage, instructions are taken from the instruction queue that instructions were fetched into, and sent to individual scheduling windows in a particular cluster. Scheduling refers to when the input operands for an instruction are ready and the instruction is scheduled for execution. Until scheduling, every instruction goes through the processor *in-order*, i.e., in program order. In scheduling and execution, instructions execute *out-of-order* as they are ready to execute. Later, in the retirement stage, instructions are put back in program order.

The issue() function in the GProcessor object takes a configurable number of instructions from the instruction queue and tries to put them in the scheduling

queue for each cluster. The issue() function calls addInst() for each instruction. If addInst() fails for an instruction, issue() returns, and issue() will try again in the next cycle to issue that instruction.

The addInst() function checks several things before it confirms that the instruction can be issued. First, it checks that there is space in the reorder buffer. Second, it checks that there is a free destination register. Third, it checks that there is space in the scheduling window for the cluster. Finally, based on the specific resource that an instruction uses, it performs other checks. For example, for loads, it will check that the maximum number of loads has not been reached. For branches, it will check that the maximum number of outstanding branches has not been reached. This check is done by calling the schedule() function of the specific Resource object for that kind of instruction. (Examples of Resources are load/store units or floating-point multipliers).

If the above-described addInst() function in the GProcessor class succeeds, it calls the addInst() function for the specific Cluster which will later execute the instruction. Finally, an entry is added at the tail-end of the reorder buffer for this instruction.

To manage dependencies between instructions, i.e., when the destination register of one instruction is the source register of another, each Cluster has a DepWindow (dependency window) object which manages the dependencies between instructions. The addInst() function in the Cluster then calls the addInst() function in the DepWindow (dependency window) associated with the Cluster.

In the DepWindow, a table is maintained mapping instructions to destination registers. The table only keeps track of the last instruction to write to each register. When addInst() is called, the source registers of the instruction being scheduled are looked up in the table. This finds the instructions, if any, which produce the input operands for that instruction. When an instruction has finished execution, if the entry in the table still maps to that instruction, it is cleared.

Consider the following code fragment:

```
x: ld R1, 0x1000 ; load memory address 0x1000
y: add R1, R1, R2
```

In the example, instruction y is said to be dependent upon instruction x. Instruction y is also the consumer of instruction x, and instruction x is the producer for instruction y. In this example, the addSrc() function is called on instruction x with the parameter of instruction y. Later, in the Execution subsection, we will elaborate how instruction x can *wake-up* instruction y after instruction x has executed and schedule instruction y for execution. Also, DInst y marks that it depends upon one instruction.

In the case that there are no dependencies for an instruction, the instruction is scheduled for execution. This is done by generating a callback event to execute the simTime() function of the Resource in a certain number of cycles. The delay depends upon the fixed scheduling penalty and the number of instructions that

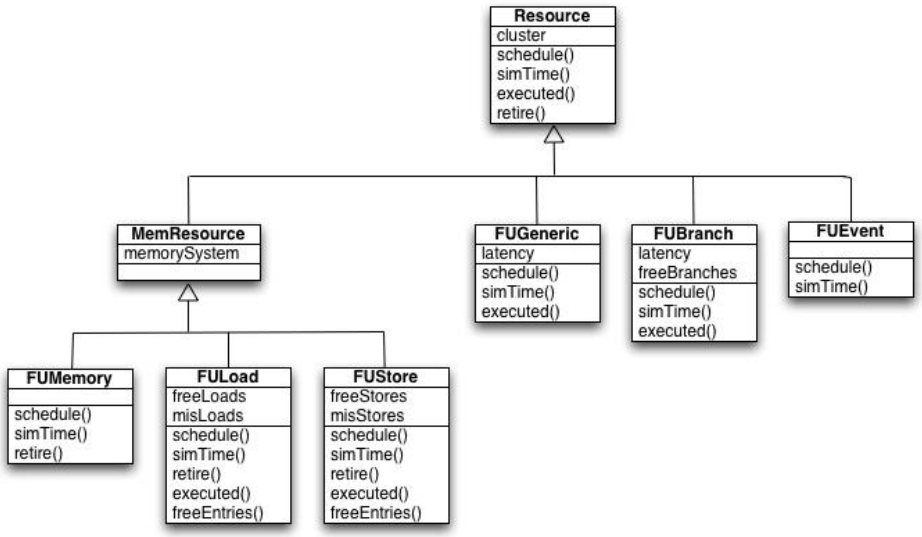

Figure 4: The Resource class hierarchy.

are ready for execution in a specific Resource, as each Resource can only execute a small fixed number of instructions per cycle.

Finally, the output operand in the table is set to point to the instruction being scheduled. Then, future instructions that consume that output operand will become dependent upon this instruction.

3.3.3 Execution

The DepWindow object either schedules an instruction for execution, or sets pointers in the instructions which must execute first.

There are subclasses to Resource for each type of instruction, as shown in Figure 4. Loads are processed by the FULoad subtype, stores by FUStore, other miscellaneous memory accesses by FUMemory, branches by FUBranch, and all others by FUGeneric. Each subclass defines a simTime() function which simulates the execution of the function. Each also defines an executed() function, which is called after execution has completed. Execution takes one cycle for branches and integer operations, several cycles for floating-point operations, and up to hundreds of cycles for memory operations.

Execution is done by the simTime() function in the Resource object. For loads, the load Resource object subtype, FULoad, will send the load to the cache. For stores, it schedules a callback to the executed() function. (Stores are actually sent to the cache in the retirement stage, since at this stage, the stores could be down the wrong path of a branch.) For all other instructions, a callback is scheduled to the executed() function.

The executed() function calls the entryExecuted() function for the corresponding Cluster for that Resource(). The entryExecuted() function calls the simTimeAck() function in the DepWindow object. The simTimeAck() function first marks the DInst as executed, so it can be retired later. Then, it checks if the entry for the destination register in the table still points to this DInst. If so, it clears that entry in the table; future instructions which consume that value do not need to wait. Finally, the instruction checks if there are any instructions dependent upon it. Consider again the example above. In this case, DInst x checks that DInst y is not dependent upon any other instructions. Then, y is scheduled for execution by generating a callback event to the simTime() function.

3.3.4 Retirement

In the retire stage in an out-of-order processor pipeline, instructions which have executed are removed from the head of the reorder buffer in the original program order. Typically, up to 3 or 4 instructions can be retired per cycle. Most instructions can always be retired once they have executed and have reached the head of the reorder buffer. Stores, however, are not actually sent to the cache until the retirement stage. If the cache cannot accept the store right then, the store cannot be retired.

In SESC, the GProcessor object calls the retire() function each cycle. The retire function takes the DInst at the head of the reorder buffer, which is the oldest instruction, and checks if it can be retired. First, it checks if the instruction has been executed by checking the executed flag set in the execution stage. Then, it calls the retire() function for the Resource object that handled that instruction.

The retire() function for the Resource object returns a success code indicating whether or not the instruction can be retired. For all instructions but stores, the instruction can always be retired. For stores, the retire() function checks if the cache can accept the store and then returns the corresponding code. The cache might not be able to accept the store if there are too many outstanding stores or if there is a memory fence. If the cache can accept the store, a memory request is sent to the cache for that store and the instruction can be retired.

The retire() function in the Resource object also is responsible for destroying the DInst object.

In the GProcessor object, the retire() function is a good place to check when the processor has stalled for too many instructions. (I.e., when the processor is no longer retiring instructions.) If a processor stalls too long, it likely indicates the presence of a bug.

3.4 Caches

In a typical modern micro-processor, there is an upper-level cache for instructions, an I-cache, and an upper-level cache for data, a D-Cache. These are also

referred to as L1 caches. Below this is a larger, slower L2 cache. In many configurations, there is also an off-die L3 cache that is even larger and slower than the L2 cache. Caches have many parameters. SESC models different cache:

- Sizes
- Hit & Miss latencies
- Replacement policies
- Cache-line sizes
- Associativities

The Cache implementation in SESC is necessarily quite complex and uses many event-driven callbacks. This is necessary to model all the latencies and transactions involved in caches. (For example, a read miss in the L1 may cause a dirty cache line in the L1 to be written back to the L2, and only after this is done can the L1 miss be sent to the L2. Then if the read misses in the L2 cache, the L2 has to arbitrate for access to the bus, then send it to memory, and so on. Each of these steps incurs a fixed delay plus the cost of arbitrating with other requests for fixed resources.) In this section, we will describe only the most important parts of the Cache implementation, as the details would otherwise be overwhelming.

Each GProcessor has a MemorySystem object. The MemorySystem object creates the hierarchy of caches and serves as an adapter between the GProcessor and the highest-level Caches.

When the GProcessor needs to interact with the MemorySystem, it creates a MemoryRequest object. There are DMemoryRequest objects for data and IMemoryRequests for instructions. Further, DMemoryRequests have a field to indicate if an access is a read or a write. MemoryRequests are created through the singleton DMemRequest::create() or IMemRequest::create() functions. These functions take as parameters the DInst object the request is associated with, the MemorySystem object this request is going through, and the type of operation (read or write).

The create() function allocates a new MemRequest object, initializes it, and then calls the access() function on the highest-level Cache object.

In the Cache object, the access() function is quite simple. It sends read requests to the read() function and writes to the write() function. Each of those functions uses a callback to call doRead() or doWrite() in the next available Cache cycle, to model contention for the cache ports. If the read or write is a hit in doRead() or doWrite() respectively, the goUp() function is called on the MemoryRequest. This will return the MemoryRequest to the GProcessor. Otherwise, a miss handler function is called which calls the goDown() function on the MemoryRequest to send the MemoryRequest to the next level in the hierarchy (e.g.,the bus or the L3 cache).

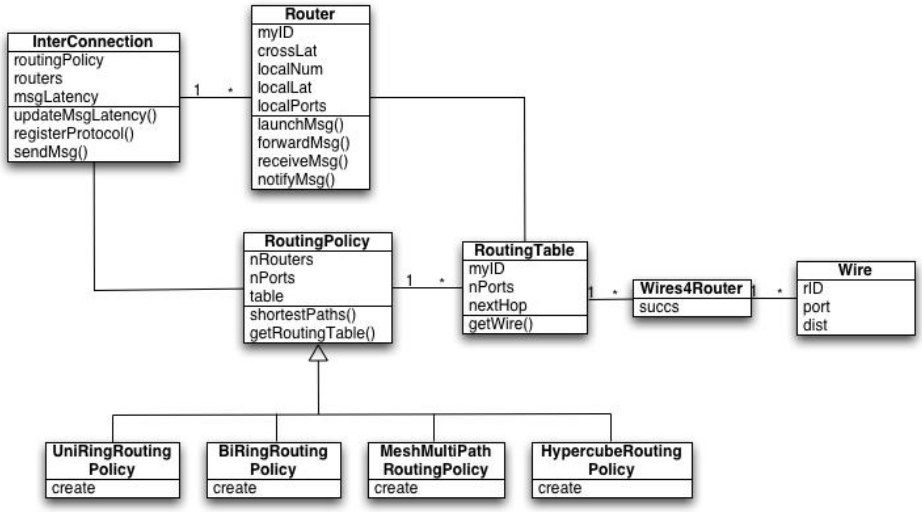

Figure 5: Interconnection network class organization.

The important thing is that all types of Caches and Buses inherit from a common class, MemObj, which defines a common interface consisting of access(), returnAccess(), and other less important functions. (ReturnAccess() is called by goUp() when an access returns to a higher-level cache from a lower-level cache.) This common interface allows fully-configurable cache hierarchies. Each Cache subtype can have a different manner of handling requests internally, as long as it conforms to this interface to upper and lower-level caches.

In a multiprocessor system, at the lowest level, each processor's caches and the main memory are connected. The manner in which they are connected is described now.

3.5 Interconnection network

The interconnection network refers to the communication channel between processors in a large multiprocessor system. Examples are a bus or a hyper-cube.

The job of an interconnection network in a parallel machine is to transfer data from any source node to any desired destination node. The network is composed of switches that route the packages from the source to the target. Each network node contains a routing table, which stores network path and status information and is used to select the most appropriate route to forward the packages along.

Figure 5 shows a UML representation of the classes that compose the network module. The InterConnection class represents the whole network layout. An InterConnection object is defined by two components:

- A set of Router objects. The Router class represents a router in an in-

terconnection network. It decides where to send the packages it receives according to the routing table and the ports traffic flow. Each Router is defined by and ID and a set of parameters that model the dynamic behavior of the switch:

  - crossLat: router crossing latency
  - localLat: local port latency
  - localNum: number of addressable local ports
  - localPort: number of ports for each addressable local port

  A message is injected in the network via the launchMsg function. Messages are sent from router to router using the forwardMsg function. E.g., if a message has to go through five routers, forwardMsg is called five times. Once a message arrives at its destination, receiveMsg is invoked. Each router in the network has its own routing table, represented by the RoutingTable class. The RoutingTable has the Wire objects that connect routers to each other. The Wire class represents an unidirectional link between two routers.

- A RoutingPolicy object. This abstract class is in charge of building the routing tables for a given network configuration. It has four descendants:
  - FullyConnectedRoutingPolicy: fully-connected network
  - UniRingRoutingPolicy: unidirectional ring
  - BiRingRoutingPolicy: bidirectional ring
  - HypercubeRoutingPolicy: hypercube mesh

3.6 System Calls

SESC does not provide an operating system, so SESC must trap system calls and perform them on behalf of the application. For every standard system call, SESC transforms it into a MINT function. For example, the function call `lstat` is redirected to the MINT function `mint_lstat`. For a complete list of system calls and their corresponding MINT function, please refer to the file subst.cpp.

The emulator, MINT, simulates most system calls. However, there are some system calls that are not simulated. For example, fork() and sproc() need to interact with the the operating system because it is not clear how the newly created process will be scheduled. Since SESC does not provide an operating system, libapp is the application interface that takes care of those kind of functions.

The other responsibility of libapp is to emulate Pthreads. The Pthreads interface was specified by the IEEE POSIX 1003.1c standard (1995), and there is vast documentation on it. When compiling applications for SESC, the developer can choose between compiling an application to run under SESC or to run natively on the host computer. The Thread API is the same in both cases, but

if the developer decides to use the OS-native thread system, this API is just a wrapper for the host's pthreads library. Finally, libapp is also responsible for the locking API.

3.6.1 Thread API

- `void sesc_init()`: Initializes the library and spawns the internal scheduler thread and transforms the single execution unit of the current process into a thread. It must be the first function call of the thread API that is called from an application.
- `sesc_spawn(void (*start_routine) (void *),void *arg,long flags)`: Creates a new thread of control that executes concurrently with the calling thread. The new thread calls the function `start_routine` passing `arg` to it as the first argument. It returns an unique thread ID. For more information about the flags, please check sescapi.h.
- `void sesc_wait()`. Blocks until one of the child threads have finished. If there are no child threads, it returns automatically. Unlike traditional `wait()` calls, it does not return the `pid` of the thread that finished.
- `void sesc_self()`. Returns the current thread ID.
- `int sesc_suspend(int pid)`. Suspends a thread whose pid is equal to the argument. The thread is transitioned to suspended state and it is removed from the instruction execution loop. The function returns `true` on success and `false` on errors. If the calling thread is the only one in running state, the simulation concludes.
- `int sesc_resume(int pid)`. Resumes a thread in suspended state by moving it to a running queue. A resumed thread is usually assigned to the same CPU that it was running on before it was suspended in order to minimize the number of cache misses. However, the flags specified in thread creation can change this policy. The function returns `true` on success and `false` on errors.
- `int sesc_yied(int pid)`. Explicitly yields execution control to the thread whose pid is passed as a parameter. If `pid = -1`, any thread can be dispatched. The function returns `true` when it succeeds or `false` when the pid specifies an invalid or not yet ready thread.
- `void sesc_exit()`. Terminates the current thread.

3.6.2 Synchronization API

The developer can choose between compiling an application to run under SESC or to run natively on the host computer. The Synchronization API is the same in both cases, but if the developer decides to use the native-OS locking system, this API is just a wrapper for the host locking system.

- `void sesc_lock_init(slock_t *lock)`: Initializes the lock `lock`.
- `void sesc_lock(slock_t *lock)`: Acquires the lock `lock`.
- `void sesc_unlock(slock_t *lock)`: Releases the lock `lock`.

Other synchronization primitives supported by SESC are barriers and semaphores:

- `void sesc_barrier_init(sbarrier_t *barr)`: Initializes a barrier `barr`.
- `void sesc_barrier(sbarrier_t *barr, long num_proc)`: Executes a barrier `barr`.
- `void sesc_sema_init(ssema_t *sema, int initValue)`: Initializes the semaphore `sema`.
- `void sesc_psema(ssema_t *sema)`: Signals the semaphore `sema`.
- `void sesc_vsema(ssema_t *sema)`: Waits for the signal for the semaphore `sema`.

3.7 Build strategy

For performance reasons, many configuration options are chosen at compile-time rather than run-time. For the most part, quantifiable options, such as processor speed or cache size, are specified at run-time in a configuration file. Qualitative options, such as whether cache-coherence is enabled, are usually specified at compile-time.

In addition, some options are mutually-exclusive and will not compile if both options are enabled. For example, there are two different cache-coherence implementations that cannot both be used at the same time.

Most architectural proposals have a large quantitative evaluation. In the evaluation section of a paper, the numerical quantities, such as the processor speed and pipeline configuration, are usually kept the same, whereas the qualitative properties are varied. Usually a user will compile a SESC binary with their proposal enabled and a baseline SESC binary without their proposal and compare the performance results.

To enable the user to easily compile multiple version of SESC using the same source code tree, the build infrastructure stores all the object files and the final binary in a separate build directory. Thus, the user can have several different build directories with different versions of the simulator.

There is a configure script in the root directory which is used to choose which features should be enabled.

It is very simple to build the simulator. For example, to model power:

```
mkdir build_power
cd build_power
../configure --enable-power
make
```

In this example, all intermediate compilation files and the final binary will be stored in the build_power directory, and other versions of the simulator can built in other directories without interfering with each other.

4 Common patterns

In this section, some of the common objects and patterns that are used throughout the simulator are presented.

4.1 CallBack

The core of the processor is modeled as in an execution-driven simulator. In other words, functions are called to simulate some parts of the processor every cycle. The rest works as an event-driven simulator. The CallBack class and its subclasses let the programmer schedule the invocation of a function call at a given time in the future. Before describing the parent class in more detail, it is very helpful to see an example with code extracted from Cache.h and Cache.cpp.

Suppose one wants to call the `doReadCB` member function of the `Cache` class using the Callback infrastructure. One should write the following code:

```
typedef CallbackMember1<Cache, MemRequest *, &Cache::doRead> doReadCB;
```

This defines `doReadCB`, that will call the `doRead` member function of the `Cache` class. `CallbackMember1` indicates that the function we are about to call is a member function of a class and that it will receive only one parameter. The arguments of the template indicate the class to which the function we are calling belongs (`Cache`), the kind of argument the function is expecting (`MemRequest *`) and the address of the function to call (`&Cache::doRead`).

There are three operations one can do with this callback function:

- Execute it right away.
- Execute it after a certain number of cycles from now. For example, one can schedule a function to be called after 30 cycles from now.
- Execute it at an absolute moment in the future. For example, one can schedule a function to be called during cycle 100,000,000.

In the example, the callback function is going to be executed at a certain point in the future (in the object's next scheduling slot, to be more precise):

```
doReadCB::scheduleAbs(nextSlot(), this, mreq);
```

As one would expect, there are many subclasses that derive from `CallbackBase`, which in turn derives from `EventScheduler`. `EventScheduler` is an abstract class that implements two of the three functions mentioned above and declares the third as a virtual function:

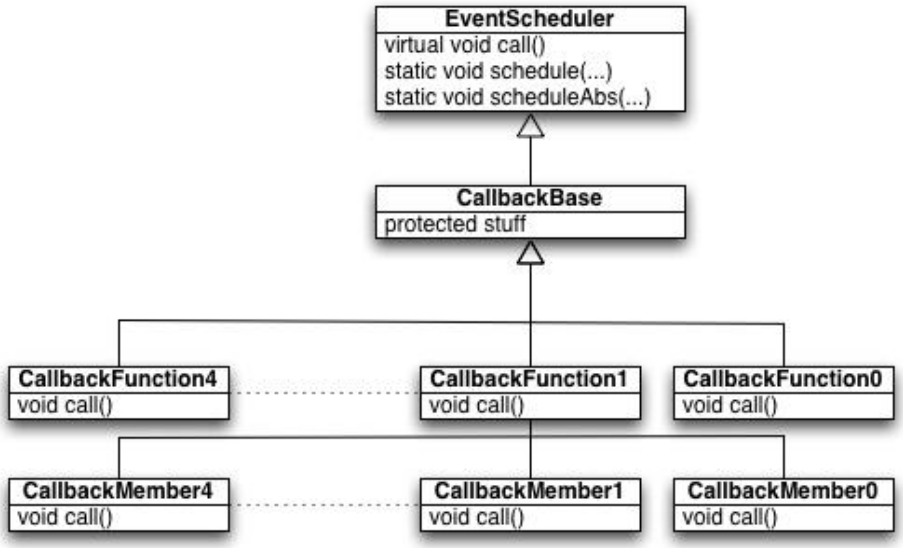

Figure 6: The callback class hierarchy.

- `virtual void call()=0;`. Each concrete class will implement it.
- `static void schedule(TimeDelta_t delta, EventScheduler *cb)`: schedules the function call to be executed after `delta` cycles.
- `static void scheduleAbs(Time_t tim, EventScheduler *cb)`: schedules the function call to be executed at time `tim`.

Figure 6 shows a simplified UML diagram of these classes. `CallbackBase` is an abstract class that provides some useful private functions, but the developer should never use it unless he is creating a new descendant. The concrete classes `CallbackFunction0`, `CallbackFunction1`, `CallbackFunction2`, `CallbackFunction3` and `CallbackFunction4` are used to call functions (outside classes) with 0, 1, 2, 3 and 4 arguments, respectively. The concrete classes `CallbackMember0`, `CallbackMember1`, `CallbackMember2`, `CallbackMember3` and `CallbackMember4` are used to call class member functions with 0, 1, 2, 3 and 4 arguments, respectively.

4.2 GStats

Any simulator needs an infrastructure to report the many statistics that a user might be interested in, and the `GStats` class and its descendants provide the developer with them.

In addition to providing a common interface to its descendants, `GStats` is responsible for maintaining a list with all the GStats-derived objects that have

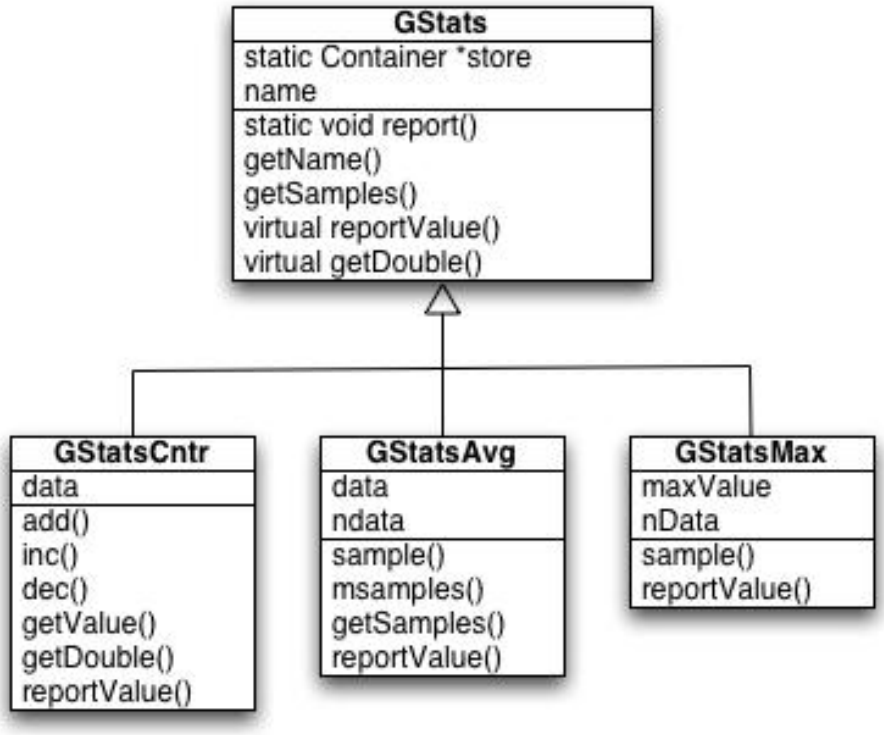

Figure 7: The GStats class hierarchy.

been instantiated. Every `GStats` object *subscribes* itself to that global list of statistics in its constructor.

Three classes derive from `GStats`:

- `GStatsCntr`: A simple counter.
- `GStatsAvg`: An average counter.
- `GStatsMax`: Stores the max value of all the given values.

All of them must implement the `reportValue()` function, which prints the value it stores or calculates. `reportValue()` will be called from the `GStats` static `report()` function, which traverses the global list of `GStats` objects.

Figure 7 shows a simplified UML diagram of the GStats hierarchy.

4.3 Pools

Some objects in SESC are allocated once and used until the simulation is finished, e.g., Processor, Resource, and Cache objects. Others have short lifetimes,

e.g., DInst and MemRequest objects. Using new or malloc() for frequently allocated and deallocated objects results in heap fragmentation and makes debugging memory leaks difficult. To enhance performance and testability, pools are used. Pools are large blocks of memory which contain objects of one type. Each dynamic object in SESC has its own pool from which objects are allocated and deallocated.

Use example in C++:

```
class Square {
    static Pool<Square> pool;
    ...
}

Square s = Square::pool->checkOut();
s.doStuff();
Square::pool->checkIn(s);
```

4.4 Debugging

To support debugging, SESC has a debugging compilation option, DEBUG. When DEBUG is turned on, assertions are checked and logging messages are output. It is important to have this as an option, since debug mode slows down the simulation considerably. SESC has a small file which defines the debugging functions.

The I(x) function treats x as an invariant expression which must evaluate to true. There are other similar functions which take different numbers of arguments.

To log a debug message, programmers use the MSG() function the same way they would use printf(). It takes a parameterized string and a list of parameters. By default, these messages are displayed to the user's shell, but they can also be redirected to a file. This is similar to the Diagnostic Logger pattern presented in Neil Harrison's paper on logging.

Use example:

```
I(x>y);
MSG("x=%d,y=%d",x,y);
```

References

Culler et al, *Parallel Computer Architecture: a Hardware/Software Approach*, 1998.

Harrison, Neil, "Patterns for logging diagnostic messages," *Design patterns in communications software*, pp. 173-185, 2001.

Hennessy and Patterson, *Computer Architecture: A Quantitative Approach*, 3rd ed., 2003.