Archsim structure

Igor Benko

UCIB #: 2005-ib01

September 2005

Abstract

Archsim is an event-based behavioral simulator intended for architectural explorations for Fleet [**?**] structures. This document describes the simulation engine and the base building blocks used in Archsim.

References

[1] Coates, Lexau, Jones: FLEETzero: An asynchronous switching experiment. SML 2000-0768

[2] Gamma et al: Design patterns, Addison-Wesley, 1995

[3] Gonnet, Yates: Handbook of algorithms and data structures, Addison-Wesley, 1991

UC Berkeley Computer Science

This document is a product of a collaboration between Sun Microsystems and the University of California at Berkeley. The ideas contained herein are freely available for any academic purpose.

1 Introduction

The Fleet paper[**?**] presents s a high-throughput asynchronous switch fabric and proposes use the switch fabric at the core of a processing unit where the core concept is that of a data move. That is, an operation on a datum takes place as a side-effect of moving the datum to a functional unit that is responsible for performing the desired operation.

The concepts presented in [**?**] leave lots of room for exploration: Can the tree-based switch fabric be competitive in terms of both speed and energy efficiency? What kind of bypasses should be added to the switch fabric? What kind of ordering must be preserved among the instructions flowing through the switch fabric? How to ensure such ordering? What kind of functional units should be attached to the switch fabric? How can one program a Fleet architecture? How can one implement branching? How can one implement multiprocessing? ...

Archsim seeks to provide a simple tool that would help in searching for answers to some of these questions. So far, the core of the simulator has been built. Lots of work still remains to be done. The most notable missing piece is an easy and preferably graphical way to assemble simulations and to review simulation results. While the base structure of Archsim is simple, this memo attempts to record common concepts, classes, and architecture to ease future browsing and understanding of the code.

2 Simulation engine

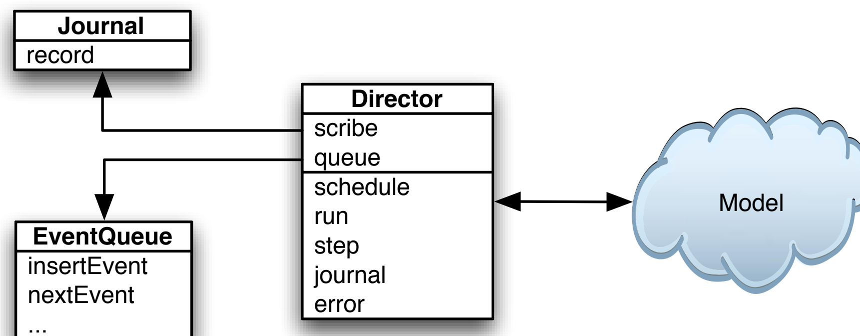

Figure 1: Archsim architecture follows the mediator pattern

Figure **??** depicts the architecture of Archism simulation engine. The mediator pattern [**?**] was the main theme used. More precisely, a singleton instance of Director acts as the mediator for a simulation. The director interacts with the event queue to fetch, one at a time, events in the order they must be simulated. How do events make it to the queue? Execution of an event may lead to future events that must be simulated. In such a case, the model invokes the director and requests that events be scheduled for simulation. Initial events are scheduled in the initialization of the simulation. Typically, these initial events correspond to the initial stimulus that is provided to the subject of simulation.

Note that the architecture does not define the structure of the simulation model. In fact, we have developed two simulation models in the process of testing the simulator. The only assumption made is that model, or its building blocks, interact with the director to schedule events to simulate.

The director provides access to functionality required to construct a journal of the simulation. At the moment, the model, or its components, ask the director to journal a specific string. I expect that this functionality will evolve when more complex types of journal are required. For example, we will likely want the simulator to produce a journal in the Verilog format, so that the Gwiz tool may be used for visualization of simulation results.

3 Event queue

The event queue stores events and returns them in the ascending order of the time stamps that are contained within each event. EventQueue is an abstract class that defines the operations that must be provided by its implementations. The list of operations is as follows:

nextEvent returns the event from the head of the queue and removes that event from the queue.

peek returns the event from the head of the queue, but the event remains in the queue.

insertEvent inserts an event in the queue.

size returns the number of events currently in the queue.

isEmpty returns true if there are no events in the queue, false otherwise.

clear removes all events from the queue.

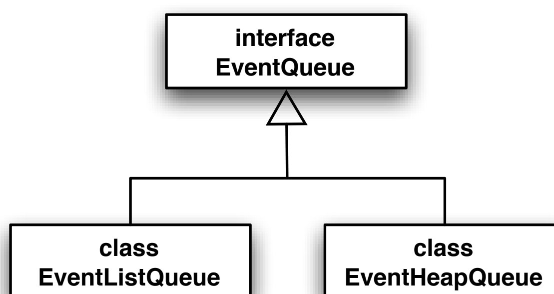

Figure 2: Event queue interface and two implementations

Currently we have two implementations of the queue. One implementation, EventListQueue is utilizes a linked list to store the events. This implementation has the property that events that are to take place at the same time are simulated in the order in which they were inserted in the queue. This property may be important when we simulate structures that involve arbitration. The cost of an insertion of an event is linear in terms of the length of the queue, and removal of the top element is a constan- cost operation.

The second implementation of the event queue utilizes a heap to store events [?]. This implementation is has a better expected performance that the linked list: Both insertion and removal of the top element are logarithmic in the length of the queue. This implementation of the queue does not preserve the order in which events were inserted in the queue.

Both implementations of the event queue can grow dynamically in size. The heap implementation uses an array to store a heap. Growth of that array is expensive because all events are copied to the newly allocated

array. For this reason, the initial capacity and the capacity increment are set to 100, and 50, respectively, with expectation that the growth operation will not be invoked frequently.

4 Journal

At the moment, only a trivial implementation of the journal is provided. The implementation simply writes journal lines directly to the screen without any additional formatting. I expect that the journal will evolve as our needs for visualization of simulation results become more clear.

5 Base library

Archsim provides a library of classes that are used in the simulation engine and also in implementations of a simulation model. Below we list and describe the most used classes comprising the framework:

Time represents time used in the EventQueue. Time objects can be compared to each other, and Delay can be added to Time, resulting in a new Time object. It is important that Time is immutable because it is used for ordering events in the EventQueue.

Delay models delays that incur in the simulated model. Note that this is a separate class from Time, primarily to reflect the difference in their semantics. Delay is also immutable - the method addDelay results in a new delay object.

Command is an abstract class that follows the command pattern from [**?**]. Commands are used to represent activity that must be carried out when an event takes place. Components in the model define their own commands.

Function is a close relative to a command. The difference is that execution of a command does not return a value, but invocation of the value method of a Function object does return a value. Functions are used to define operations for manipulation of data.

Event combines a Time object and a Command object. The Time object is used to determine when to simulate the event. The simulation of an event is then carried out by executing its command.

IDGenerator is a source of unique IDs. Note that the generator is not persistent, thus uniqueness of generated ID holds only for one simulator run. IDgenerator is not thread safe.