The Pantheon storage-system simulator

John Wilkes

Storage Systems Program
Computer Systems Laboratory
Hewlett-Packard Laboratories, Palo Alto, CA

HPL–SSP–95–14 revision 1, May 1996.

Revision history:

v0	29th December 1995	original version.
v1	17th May 1996	new runPantheon section.

This paper presents the facilities and design of the Pantheon storage system simulator. This simulator was originally intended to do performance modelling of parallel disk arrays. Over time it has been extended and generalized so that it now supports modelling of a wide range of I/O systems for both uniprocessors and parallel computers.

The intent of this document is to provide an overview of the simulator, its architecture and its capabilities. The intended audience is potential users of the simulator and people who will be assessing its results.

1 Introduction

The Pantheon simulator was designed to support the rapid exploration of design choices in storage systems and their components such as disks, tapes and array controllers. It has been functional in some form or another since summer 1992. Evidence for its usefulness include its successful application in a number of projects, such as:

- the TickerTAIP parallel RAID architecture [Cao94b]—from which the simulator got its original name (we have since rechristened it to avoid confusion);
- an analysis of idle-period detection and prediction algorithms [Golding95];
- the HP AutoRAID advanced disk array technology [Wilkes96];
- performance and availability analyses for AFRAID—a frequently redundant array of independent disks [Savage96].

This document describes the simulator and some of the philosophy behind it. The intention is to provide an introduction to Pantheon to anybody who might be contemplating using it, as well as some idea of its capabilities for people who may see output produced by it.

1.1 Document structure

The remainder of this document is organized as follows. It begins with a high-level overview of the elements of the Pantheon simulation suite: the simulator itself, the support libraries and tools, and a simple example of how Pantheon might simulate a simple disk model. It is followed by a more in-depth descriptions of the standard Pantheon simulator components, and the library components that support them. This is followed by a description of how a simulation is constructed and executed. The paper closes with some disclaimers, general comments, and a bibliography.

Several additional documents describe parts of the Pantheon support infrastructure: most of the details of these elements are deferred to those documents, rather than repeated here. As always, the source code is the definitive reference.

2 Overview of the Pantheon suite

There are two main parts to the Pantheon suite: the simulator itself, and a set of support libraries and tools that assist the generation, execution, and analysis of simulation runs.

2.1 The Pantheon simulator

The Pantheon simulator proper is constructed from a rich set of primitive simulation components (a “kit of parts”), together with infrastructure to glue these together to configure and execute a simulation. Several disk-level I/O traces are available to feed into simulations, and the package includes a set of analysis tools that can summarize the results of simulation runs.

Pantheon’s simulation modules are written in C++, compiled (optimized if you wish), and linked together to make a single Pantheon executable program. Using compiled building blocks of this form means that simulations execute at full speed: the runtime cost of linking the components together is just a C++ virtual function call. This document surveys the “standard” set that is provided as part of the Pantheon release. In addition, new components can be added, or existing ones extended, to meet particular needs.

A compiled implementation provides good runtime performance, but can be limiting in terms of configuration flexibility. To address this, Pantheon uses the interpreted language Tcl [Ousterhout94] to control which simulation modules are to be instantiated, and how they should be connected together and parameterized. Since Tcl is a full programming language, arbitrarily complicated configuration decisions are possible: for example, it is possible to calculate how many disk drives an array needs to accommodate its load as a function of its redundancy algorithms, rather than having to precompute this. The net result is that Pantheon achieves both great configuration flexibility *and* good execution-time performance.

2.2 The Pantheon libraries

Much of the Pantheon infrastructure is included in a set of libraries, structured as separate entities to allow their contents to be reused. The main ones are summarized here: more details are in section 4.

2.2.1 Raphael

Raphael is a lightweight coroutine package that supports the thread model used by Pantheon and its notion of simulated time. It is based on the QuickThreads package from the University of Washington [Keppel93b] and offers similar functionality to the tasking library [ATT89] that was used in the earlier versions of the simulator.

Code executes in a Raphael thread until either (a) it relinquishes the processor by blocking on a synchronization object such as a semaphore, or (b) it asks for simulated time to be advanced for it. There is no preemption or time-slicing. A blocked thread becomes eligible to run when the synchronization object on which it is waiting releases it. A delayed thread becomes eligible to run when the global simulation time advances to the moment it was waiting for. Simulation time advances only when there are no more threads ready to run at its current value—that is, they are all blocked or waiting for time to advance. More details are provided in a separate document on the Raphael library [Golding95b].

2.2.2 Lintel

Lintel provides a set of useful objects for handling statistics, error and result reporting, random-number generation, data structures such as queues and heaps, storage-management for small C++ objects, and so on.

2.2.3 SRTlite

The SRTlite library provides the interface to the disk I/O traces originally gathered by Chris Ruemmler and described in detail in [Ruemmler93]. The library provides facilities for merging multiple traces together, scaling and filtering traces, and a rudimentary synthetic trace-generation facility. More details are provided in separate documents on the SRTlite library and its use [Abram95, Ganger95].

In addition, a rudimentary trace replayer has been written to take the SRT traces and replay them against one or more real disks [Soepenberg95].

2.2.4 libckpt

Some simulations can run for a very long time; if a crash happens all this work can be lost. In addition, some defects only manifest themselves after a considerable amount of runtime, and it would be convenient to be able to restart a debugging run just before the bug manifests itself. To

address both of these concerns, the libckpt library allows a simulation to take checkpoints of itself, from which execution can be resumed at a later time if needed.

2.3 Pantheon support tools

The libraries provide C++ code that can be linked into the Pantheon simulator. The support tools are stand-alone programs or families of scripts that provide additional runtime support for Pantheon executions.

2.3.1 runPantheon

Although the simulator takes only Tcl expressions as arguments, these are sometimes rather unwieldy, and the runPantheon script attempts to simplify much of the complexity by providing a convenient "front end" to the simulator itself: parsing simple command-line switches, and making simplifying (but convenient) assumptions about the structure of a simulation.

2.3.2 DQS support

DQS, or Distributed Queueing System, is a simple batch-processing facility designed at Florida State University [Green93]. We use it to execute the potentially large set of simulation runs that represent a high-level experimental investigation: it farms out the simulation runs to a cluster of machines that are willing to act as processor-cycle-servers. Support for DQS is built into runPantheon (the –Q flag triggers it).

2.3.3 Adze and Tongs

A typical use of the simulator is to generate a set of simulator runs that vary parameters and workloads across the domain of interest. The Adze tools support simple numerical summarizing of the set of data that results; the Tongs tools provide a great many options for extracting and displaying the same data in graphical form. More details about the Tongs tools are provided in a separate document [Golding95a].

2.3.4 Oculus

The primary output of the simulator is one or more results files containing numerical data about the internal state of the simulated system, event frequencies, timing information, and so on. It is also sometimes useful to peek into a running simulator to watch it in action—and this is what Oculus allows. When it is enabled, a separate Tcl-based graphical viewer allows real-time visualization of a Pantheon simulation, displaying running summaries of selected statistics. More details are provided in a separate document [Aicheler96].

2.4 Pantheon in action

The intent of this section of the document is provide an overview of how a Pantheon simulation fits together, before diving into a detailed description of all its components.

About the simplest model possible is shown in Figure 1. Even in this simple a system, we can see many of the components of the simulator. Working from the top of the picture downwards, we can identify the following components:

- Experiment: controls the execution of the simulator by allowing a *batch* of work to be processed before a snapshot of the results is taken. Also decides when the run is over.
- Host: represents a processor that can inject work into the storage system.
- IOload: a thread that generates a sequence of IOitems, each of which represents a single I/O request to the storage system. By default, an open queuing model is used: I/O requests are

generated according to the timestamps in the trace file, with no waiting for a previous request to complete.

- DeviceDriver: sequences requests and sends them on to a specific storage device. (There is typically one DeviceDriver for each storage device.)
- Disk channel: a path by which requests and data are communicated between the device driver and the storage device. Paths, or Routes, are composed from a series of Links.
- Disk drive: a model of a real or potential storage device.

The next diagram (Figure 1) shows one more level of complexity: the building blocks used before are now shown to have internal fine structure, and are themselves composed of lower-level components. The additional components shown include:

- IOloadGenerator: converts SRTlite trace records into IOitems, to be returned to a calling IOload thread. In addition, the SRTlite library can optionally provide a set of trace-filtering modules (these are not shown in the figure).
- Cache: a representation of buffer space that can hold identified ranges of data (used here both in the host and in the disk drive).
- DeviceMap: essentially a demultiplexor on addressees in IOitems, a DeviceMap allows a single IOload thread to generate work for multiple device drivers.
- IOscheduler: encapsulates the policy for deciding the order in which requests should be serviced if more than one of them is outstanding at any moment. It represents both the queue and its associated dequeing policy.
- Link: a representation of a potential data-movement bottleneck. They are used here to represent the performance of the SCSI bus and the two interfaces onto it: one at the host end, one in the disk.
- Disk: models the disk controller and its policies, and acts as the connection point for all the other components of the disk drive model.

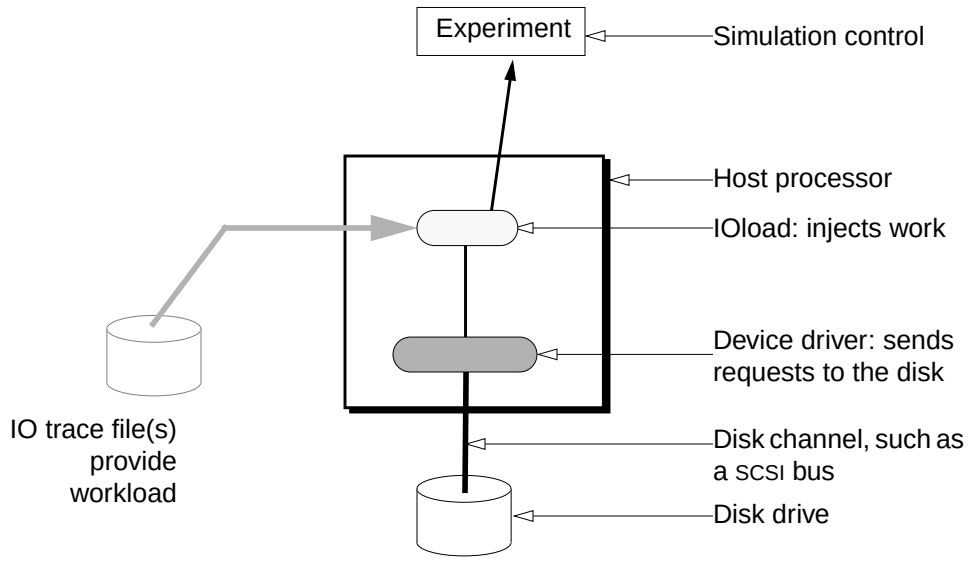


Figure 1: outline of the building blocks in a simple Pantheon simulation.

- DMAengine: represents a data pump that moves data between caches (here, between the disk speed-matching buffer and cache and the hosts's buffer cache) across a Route.
- DiskMechanism: a thread that models the moving bits of a real disk drive: the rotating platter(s) and the disk arm with one or more heads on it. Details of the disk's physical data layout such as zones, sparing, and sector sizes are handled here.

  In addition (not shown in the figure), DiskControllerData and DiskMechanismData objects are data structures that contain the values used by the Disk and DiskMechanism modules. This simplifies the support of many disk types by separating the code from the parameter data.

The first step in running a simulation is to initialize the simulator itself. This involves creating all the simulation objects desired and linking them together. (Some of them need to be explicitly linked; others automatically connect themselves.) To ensure that everything is set up correctly, semaphores are used to enable each object-initialization to run to completion, after which all threads block on an "everything is ready" event. Once everything has been created, this event is signalled, and execution commences.

The Experiment object is used to control the execution of the simulation. A typical run is divided into one or more *batches*—usually one to warm-up the simulation to a steady state, followed by one or more batches that generate results. A batch typically ends when a timer is triggered, or a set number of requests have been processed. The simulation run ends when enough batches have been run (e.g., if the Experiment decides that the results have stabilized), or if the end of one of the

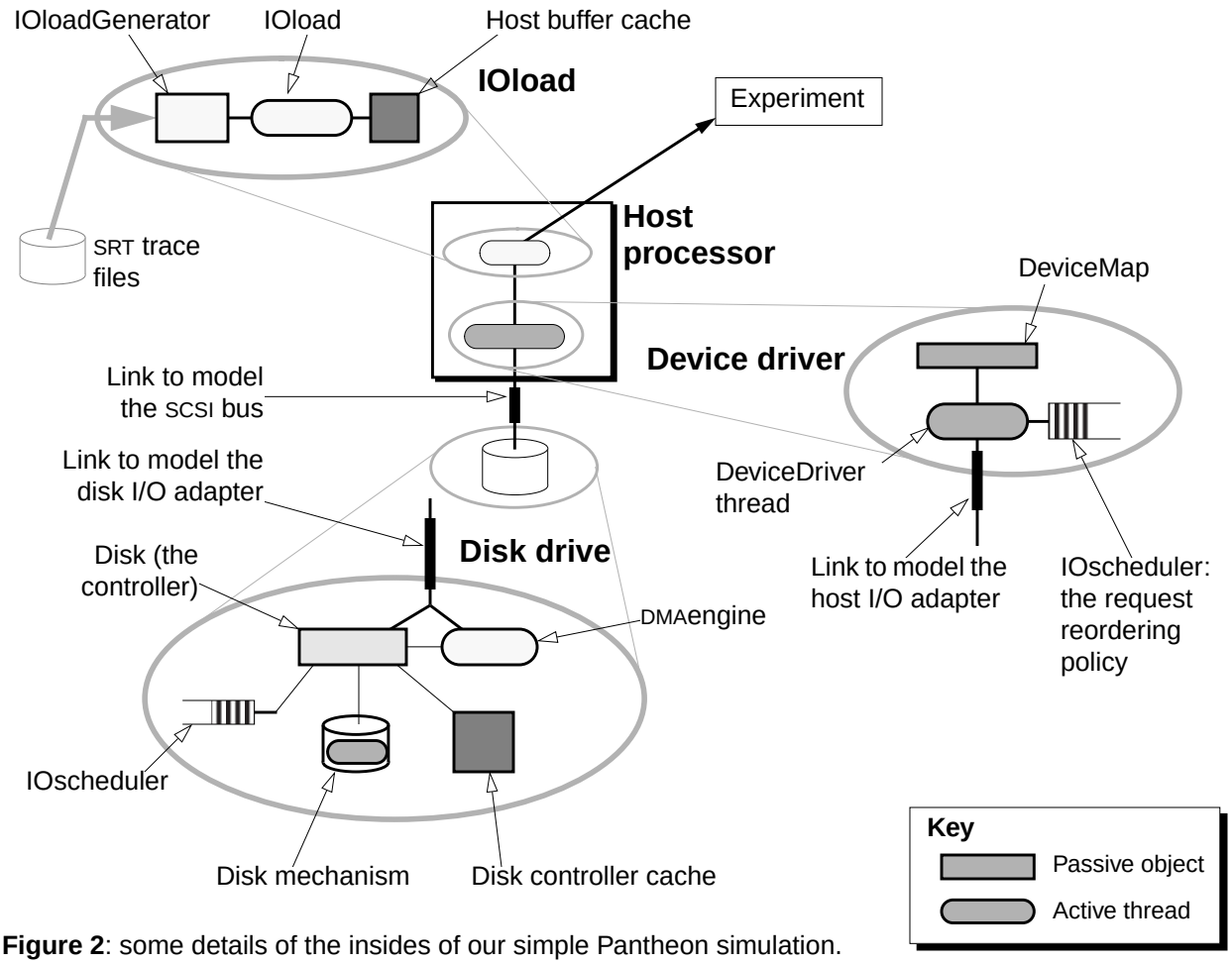


Figure 2: some details of the insides of our simple Pantheon simulation.

input trace files is reached. This technique allows separate specification of the length of the warm-up batch and of the subsequent ones, and it also keeps the workload at full steam until the last request that is to be measured has completed. Together, these eliminate start-up and termination effects, and the numbers that the simulator produces are thus those for the system's behavior in steady-state. (This technique is one of those discussed in [Pawlikowski90].)

As it runs, debugging (trace) data can be emitted by the simulator, under control of the variable DebugLevel: the higher the level, the more detail is emitted. The default IOload emits a comforting stream of periods to indicate that forward progress is being made—one for every twenty IOitems completed.

When the simulation run completes the final results are emitted. If desired, a snapshot of the results so far can also be emitted at the end of every batch: sometimes the intermediate results are useful in themselves; sometimes they provide protection against defective simulation components!

The results come from two sources: individual simulation components keep track of values they believe may be of interest, and can report them if desired. (For example, a cache buffer keeps track of how many times it is asked to provide or release space.) The majority of the results usually come from special statistics-gathering Stats objects that can be attached to measurement points throughout the simulator. The statistics gathered can be a simple count and mean (e.g., for an I/O time), or a histogram of values, or as complicated as a correlation between two values such as I/O time and request-queue length. The level of detail in the statistics can be controlled by a simple integer variable (StatsLevel). No statistics are gathered unless a Stats object has been attached to a measurement point—this means that lightweight runs pay almost no penalty, but a great deal of additional information is available if it's wanted.

The results themselves are emitted through a Reporter object, which takes care of formatting the output to correspond to the particular needs of the analysis tools. Our first Reporters were designed to generate output in a format that could be understood by programs like awk; our current ones emit Tcl scripts that are amenable to post-processing by the Tongs tools.

Work items in Pantheon are represented by structures called IOitems. These are created by IOloadGenerators, and injected into the simulator by IOloads. A single I/O request may result in more than one internal I/O action (e.g., a write to a mirrored disk will generate two low-level disk writes). In this case, the IOitem is *cloned*, and its children set to point to the original via an intermediate data structure called an IOitemVector, which is used to determine when all the child IOitems have completed. IOitems can be cloned *synchronously* (they must complete before their parent can) or *asynchronously* (the parent can complete before the clone is fully dealt with). The second is used to handle behavior like write-behind in a disk drive.

The IOitem–IOitemVector nesting can be repeated, forming a tree of linked IOitem blocks. Eventually, all the child IOitems are completed, and the parent IOitem returns to the IOload that sent it off, where it is garbage-collected. As each IOitem is completed, an opportunity exists to accumulate statistics about it—usually through a structure called an IOitemStats, which provides a raft of measurement points to which Stats objects can be attached. By this means, it is possible to accumulate I/O completion statistics at many levels throughout the simulation simultaneously.

2.5 The Tcl interface

As mentioned above, Pantheon uses a Tcl interpreter to provide a flexible language for its configuration. The interface between the Tcl code and the C++ code is provided by a set of (fairly

exciting) C++ macros, which have the effect of instantiating a set of Tcl functions that provide interfaces to a newly-created simulation object. Full details are provided in [Golding94]. Most of the time, an author of a new simulation object just uses cut-and-paste of the Tcl interface code from an existing one, so the full generality of the interface is rarely an impediment to its exploitation.

3 Details of simulation components

This section provides a more detailed description of the main simulation-component families. As mentioned above, it's always possible to add new ones, but these are common building blocks on which many other things can be constructed. A similar level of detail is provided for the Raphael, Lintel, and SRTlite libraries in the next section.

When reading the code, it may be helpful to be aware of two kinds of naming conventions:

- Class names are designed to represent the inheritance hierarchy in that the name of a more specialized class is usually the name of its base class with something appended to it. This allows the Tcl interface (section 2.5) to do a rudimentary form of type-checking: it can be set up to expect objects that are of a given class or its descendants.
- Capitalization hints are often used to distinguish different kinds of C++ names. Thus, if a name …
  - starts with a CapitalLetter: it is probably a global variable or a class name;
  - starts with a lower case letter: it is likely to be a local name, and most likely is:
    - a local variable if it contains underlines between its words like_this;
    - a member function if it uses internal capitals at word boundaries likeThis;
  - is all in capitals: it is probably a macro or #define, as in the usual C convention;
  - ends with a capital letter: it is most likely a macro argument: likeThiS;

All of the C++ scalar types defined for the simulator have #defines for MAX*typename* and BAD*typename*, representing the largest allowed value and an invalid one. Convenient types include:

- byte_t: used for byte-counts and byte lengths, such as for request sizes;
- count_t: used as a general unsigned integer;
- diskaddr_t: expresses device addresses in units of a basic unit (currently 256 bytes)—this avoids overflow of a 32-bit field for large disks and arrays;
- LintelTime: the type for simulated time (units are seconds)

3.1 I/O processing: generating and routing IOitems

An IOitem describes a single I/O request in the simulator. It specifies the request to be performed and a number of other properties of it, including:

- the operation to be performed (Read, Write, Seek, etc.);
- a logical unit number (LUN), start address and transfer length (these are encoded in a structure called a DiskRange, which IOitem inherits from);

- an identifier and subidentifier, used for debugging purposes (the subidentifier is incremented when an IOitem is cloned; the id when a completely new IOitem is created);
- a pointer to a parent IOitem and/or IOitemVector, used to wait for cloned IOitems to complete;
- a CacheRange specifying the data source (Write) or sink (Read)—more on this below—and a Route to transfer the data across;
- a priority to be used for scheduling;
- statistics, including when the IOitem was created, when useful work on it was begun, and when it finished, as well as a pointer to an IOitemStats object;
- lots of flags.

3.1.1 I/O request generation: IOloads and IOloadGenerators

An IOload is responsible for inserting work into the simulator (which includes creating or absorbing fake data space for an I/O), issuing requests to one or more DeviceDrivers through a DeviceMap, implementing the overall simulation queuing model,[1] and reporting to a controlling Experiment how many I/O requests have been handled.

IOitems are created by IOloadGenerators.[2] Two kinds of IOloadGenerators are provided: IOloadGeneratorTrace turns SRTlite trace records into IOitems; IOloadGeneratorSynthetic provides a rudimentary synthetic workload generation facility.

3.1.2 I/O request direction: DeviceMap

A DeviceMap provides a way to direct or send an IOitem to one or more storage devices. A DeviceMap has a set of "input" logical devices (LUNs), and a set of "output" physical devices pointed to by an array of DeviceDrivers. Each of the input and output devices has a size, and their concatenation forms two address spaces. The simplest use of a DeviceMap is to set the two device lists to have identical sizes for each of their elements: that is, to form a 1-to-1 map from input LUNs to output devices. More complicated uses include partitioning a single logical address space across multiple devices—which means that a single incoming request can generate multiple output requests, if it happens to span a device boundary.

3.1.3 I/O device management: DeviceDrivers and IOschedulers

A DeviceDriver sends IOitems to its associated device across a given Route (see section 3.2). It is responsible for coping with devices that can only have a fixed number of outstanding requests, and it provides a placeholder for a policy that determines the order in which requests are sent to their device.

The request-sequencing policy is manifested as an object called an IOscheduler. (This scheme of having separate policy objects that can be changed at simulation-construction time, rather than simply passing in a set of parameters to an algorithm that tries to cope with all possible combinations, is one of the more positive results of our use of Tcl). A wide variety of IOscheduler objects are provided, covering a substantial fraction of the designs reported in the literature. The most useful are probably:

- IOsched_FCFS: first-come, first-served

1. The current implementation uses open queueing. Some information on the dangers and tribulations of doing either pure open or closed queueing can be found in [Thekkath94] and [Ganger95].
2. This separation of IOload from IOloadGenerator is a historical hangover from a restriction on nested inheritance in the ATT thread library that we used to use.

- IOsched_CLOOK: bidirectional elevator algorithm that does not force a seek to the extreme ends of the disk (that policy is called CSCAN); this is probably the policy that is used in most UNIX system disk drivers;
- IOsched_ASATF: one of a family of "2-dimensional" scheduling policies that attempts to exploit information about the low-level seek and rotation-position timing inside a disk [Seltzer90b, Jacobson91].
- IOsched_prio: takes a set of IOschedulers that each correspond to a different priority, and does a strict prioritization between these levels.

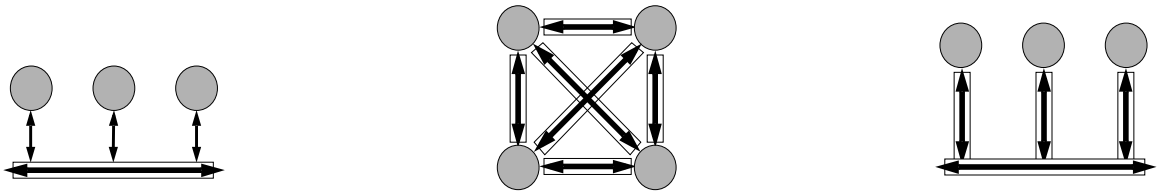
In the Tcl code that supports Pantheon, the functions create_IOsched and create_device_driver remove much of the drudgery associated with constructing all the bits and pieces for these components, including statistics objects (see section 4.2.1).

3.2 Interconnects: Links, Routes, and DMAengines

An interconnect is something that can deliver a *message*—a vector of bytes—from a source to a destination.

The primitive interconnect component is called a Link: an interconnect is made up of a set of them connected together in a suitable way (see Figure 3 for some examples). A Link has a *bandwidth* (possibly infinite), a *start-up overhead* (possibly zero), and a *transfer granularity*, representing the units in which data is moved across it. A Link also represents a point of contention that can be competed for by threads—the number of simultaneous transfers supported by a Link can be specified (it defaults to one). LinkMultiDir and LinkFibre support different bandwidths in different data-transfer directions, and were designed to model things like different service classes across a FibreChannel fabric. A zero bandwidth, zero-overhead Link can be used simply to glue components together.

Messages are sent across Routes, which are simply sequences of Links that need to be traversed. To transfer a message, a Route must first be *claimed*, which in turn calls claim on all the Links in it. Claiming the Route costs simulated time equal to the sum of the Link overheads. Data can then be moved across the Route, at the lowest bandwidth of any of the Links in it, after which the Route should be released, allowing other transfers to occur across it. Deadlock is avoided by sorting the Links in a Route and claiming them in that order. (Although this doesn't perfectly model the effects of contention inside complex meshes, it's a reasonable approximation for much I/O work.) The first and last elements of a Route are often Links representing the DMAengines at hosts or storage devices.



a. *bus*: a single simplex link, shared by multiple clients.

b. *point-to-point*: one link between each pair of communicating entities

c. *mesh*: Links at each node model the single-node send/receive rate, and a single multiplex link approximates the switching fabric

Figure 3: some sample interconnect topologies constructed from Links.

DMAengines represent data pumps that move messages across Routes. They process data-movement requests serially: if you want parallel data movements, you need multiple DMAengines. Data movement requests are specified by DMArequest structures, which include a source, a destination, and a Route, as well as some additional flags. Since the sources and destinations are described in terms of Caches (of which more below), the data movement will block if data is not yet ready at the source, or if space is not yet available at the destination. By default, if a transfer blocks in this fashion it will relinquish the Route, but this can be disabled if desired.

3.3 Processors

Processing elements are modelled by Cpu objects, which allow simulated processes to compete for cycles. The mechanism is quite simple: cycle consumers serialize on the Cpu resource, and then delay for either a given number of processor cycles or a specific elapsed time.

3.4 Caches

Caches model buffer space in Pantheon. They include support for a simple speed-matching pipeline or FIFO buffer, caches that can hold just one contiguous thing or multiple separate items, multi-segment caches, and ones that have arbitrary replacement policies like LRU. The Cache hierarchy looks like this:

Cache — virtual class that defines common operations
  CacheSpace — a single segment of memory
    CacheSpaceNoOverlap — restricted so that its contents cannot overlap each other
      CacheSpaceReplacement — adds a CacheRange-discard policy such as LRU
  CacheSegmented — multiple segments, each a CacheSpace
    CacheSegmentedNoOverlap — restricted so that contents may not overlap
      CacheSegmentedReplacement — adds a segment-choice policy such as LRU

A Cache is simply a virtual class that specifies the operations that can be done on all the different kinds of cache: adding and removing address-ranges, searching for things that overlap a given range, and so on. A Cache can be a single CacheSpace, which is the simplest object that can store data, or a set of CacheSpaces with some code to decide which one gets used for what—this is known as a CacheSegmented. Typically, CacheSegmenteds will have only one contiguous range of data in each of its segments: they are primarily designed to model the segmented caches that appear in disk drive controllers.

Associated with a CacheSpace is a Buffer object that represents a pool of reservable memory that backs the Cache; Buffers are divided up into fixed-sized *slots*, which are the unit of space reservation. A Buffer can be infinite in size: if it is not, a request for space will block until there is enough free memory available for the caller's request to be granted in toto. (Warning: the code for deadlock avoidance in cache-space management is one of the hairier pieces of design in the simulator; you can find it in CacheSpaceReplacement::claimSpace.)

3.4.1 CacheRange

Caches are placeholders for stored items. To create such a stored item, a DiskRange is *inserted* into a Cache, and one or more CacheRanges are created. A CacheRange has several portions, each represented by a DiskRange, with a LUN, start-address, and length:

- range: a portion of the logical address space that the CacheRange backs;
- space: the logical amount of buffer space that it holds; this starts out at zero until the CacheRange is *populated*;

- buffer: the physical amount of memory reserved for it by the underlying Buffer (this can be larger than the space if the Buffer has large slots);
- valid: the amount of the space that holds valid data (e.g., stuff that has actually been read in off a disk platter);
- ready: a subset of the valid portion may be ready and available for consumption (e.g., by a DMAengine).

In addition, there are producer-consumer Semaphores, and flags indicating whether the CacheRange holds *dirty* data that has yet to be written out, or if it is *pinned* in memory and must not be selected for deletion by one of the replacement policies described below.

There are two basic ways in which CacheRanges are used:

- In speed-matching buffers between a producer and a consumer (e.g., a DMAengine and a DiskMechanism). Here, the producer extends the range, space, valid, and ready areas while the consumer shrinks them. Extensions occur at the ends—i.e., the lengths increase—while shrinkage occurs at the beginning of the ranges, by advancing their start addresses (and adjusting the remaining lengths accordingly). DMArequests with the consume_source flag set trigger this behavior.
- In buffer caches, that have long-term state. Here the typical mode of use is that the range and space lengths are set equal when the CacheRange is created, and the valid and ready lengths advance in lock-step. The start addresses of these elements never change: instead, the CacheRange is deleted when it is done with.

There are circumstances where a thread needs to await some set of states across multiple caches—for example, that one cache contains no dirty data that overlaps a given range, while the other contains no pinned CacheRanges. Unfortunately, in the time required for the second set of conditions to become true, the first set might have been invalidated. The CacheLookupAndWait functions address this by backtracking to reestablish earlier conditions if necessary.

3.4.2 CacheRangeList

Some kinds of cache (notably the ones that have replacement policies) generate many fixed-size CacheRanges if a large DiskRange is inserted into them. To make life easier, a CacheRangeList is used to collect these together. It is just what it's name implies: a list of pointers to CacheRanges. Note that a CacheRange has an independent existence from a CacheRangeList; indeed, a CacheRange can be on several CacheRangeLists at the same time. This means that deleting a CacheRangeList has no effect on the underlying CacheRanges that it points to. By convention, the CacheRanges on a CacheRangeList are held in increasing address order.

A special kind of CacheRangeList (a CacheRangeListHash) is used inside Cache…NoOverlap and Cache…Replacement to allow fast lookups: because these caches do not allow their CacheRanges to overlap one another, the result of a probe for a given <LUN, start-address> pair will always be at most one CacheRange, so a hash-based search can be used.

3.4.3 Replacement and Flush policies

A CacheSpaceReplacement is a CacheSpace that has a policy for deciding what to do if a new CacheRange is to be created and populated, but there is no more space to accommodate it. In this case, a Replacement policy is called to decide which existing CacheRange(s) to evict.

If a Replacement policy (or anything else for that matter) picks a dirty CacheRange to evict, it will usually first have to be written out to a lower-level device. The Flush policy allows additional

things to be written out at the same time—for example, the FlushAggreggate policy will write out any dirty CacheRanges that form a contiguous chunk with the selected CacheRange.

By analogy, a SegmentedReplacement policy is used to decide which segment in a CacheSegmentedReplacement should be emptied when a new segment is required. (No separate FlushSegmented policy is needed.)

3.5 Disks: disk mechanisms and controllers

A disk drive is represented by a number of components:

- a Disk thread to model the controller policies;
- a DiskMechanism thread to model the mechanical parts—platters and heads—as well as data layout; it services requests one at a time, to completion;
- two IOschedulers: one to model the request-ordering policy at the controller if multiple outstanding requests are allowed, the other to sequence requests between the controller and the DiskMechanism;
- a Cache to model the speed-matching, prefetching, and write-behind buffer on the disk controller;
- one or more DMAengines, to model the data pump behind the I/O interface;
- a Link to represent the performance bottleneck of the I/O interface itself;
- associated with the Disk is a DiskControllerData structure that provides the parameters that describe the controller's behavior; a similar DiskMechanismData structure exists for the DiskMechanism; together, these separate the code fairly cleanly from the parameter values, and make creating Disk and DiskMechanism objects much simpler.

When an IOitem comes into a Disk from a DeviceDriver it is first put on the controller's IOscheduler queue (so it can be resequenced if needed), and then an attempt is made to handle it if the disk is not already handling its maximum number of concurrent requests.[1] Handling a request involves making space for it in the cache, generating a request for the DiskMechanism if needed, and creating and queueing a data-movement request for a DMAengine. The DiskMechanism may have a readahead request that it is processing: this has to be waited for, and no new ones generated, if the mechanism is now needed to do work for the new request.

Much of the complication comes from managing the cache. Reads that hit in the read-ahead buffer and writes that can fit into the immediate-report write buffer or NVRAM cache can be serviced without waiting for the mechanism; all others require the disk mechanism to be involved for part or all of the data.

The following discussion of how I/O requests are handled assumes a single-segment cache and at most one outstanding active request at the disk for simplicity. For *reads*:

1. If all of the data requested is already in the cache (e.g., because of a read-ahead), the mechanism is not disturbed. Any data in the cache that precedes the start of this request is discarded, to allow maximum space for the read-ahead if it is still continuing or will be restarted.
2. If no part of the data requested is in the cache, then the cache is emptied, a new entry inserted into it, and a request is sent to the DiskMechanism to fill it out.

[1] There is an additional policy question to do with overlapping requests that complicates this a little: the collisionQueue structures exist to resolve this question.

3. If some of the data requested is in the cache, anything preceding its start is discarded, and a request sent to the DiskMechanism for the remainder.

In all cases, a request is generated for the DMAengine to move data from the cache to the host, and the read finishes when this transfer has completed. Once the mechanism has finished its work, the controller may choose to give it one or more readahead requests—either until its cache buffer fills up, or a new request arrives from the DeviceDriver.

Writes in some disks can be immediately reported if the IOitem flag allowing this is set and it is not too large (see [Ruemmler93] for more details). Thus:

1. If the write can be immediately reported, space is made for it in the cache, the incoming DMAengine is started, and a request dispatched to the DiskMechanism to transfer the data to the disk proper. The request completes when the incoming DMA transfer is done.
2. If the write cannot be buffered, the only difference is that the request is not considered complete until the DiskMechanism has finished its write to disk.

3.6 DiskSimple and ArraySimple

The DiskSimple family is designed to support fast, crude approximations of real disks: its members do none of the fine-grained modelling that has been described above, but instead use random-number generators to estimate disk timing information. Because their running time is much less than the full disk model they are often helpful during the debugging of a large simulation.

The ArraySimple class models a (very simplistic) RAID 5 disk array. It uses the DiskSimple class to generate internal timings.

4 Details of support libraries

4.1 Raphael

Raphael provides the coroutines on which Pantheon's pseudo-multithreading is built. In addition, it provides a notion of simulated time, as well as a suite of synchronization objects for various purposes. Many more details are provided in [Golding95b].

A Raphael thread is used to model asynchronous processing in Pantheon. Once instantiated and initiated, each thread runs until it is blocked by waiting for an event or by delaying, which means waiting for global simulation time to advance. If no threads are ready to run at the current global simulated time, this time is advanced to the point where the first delayed thread can run.

Threads can block on Blockables, which come in many flavors. Examples include:

- Barrier: a thread waiting on a Barrier is stuck there until another thread calls signal() on the Barrier to make it ready.
- Semaphores: a Mutex is a binary semaphore; a plain Semaphore is a counting semaphore that increments and decrements by one; a SemaphoreMulti allows atomic multi-unit increment and decrements.
- Locks: these collect together a set of LockObjects and allow construction of things like lock-matrices (e.g., for single-writer, multi-reader protocols).
- Future: a placeholder for a result from an asynchronous operation. When the result is computed, any thread waiting for the result is allowed to proceed.

- Message queue: a FIFO-ordered set of objects: a thread can wait for something to be put onto a MsgQueue (although most commonly used for IOitems, they can be used for anything since the class is a template);
- Timer: a kind of Barrier that supports waiting for a timeout.
- BlockableSet: lets a thread wait for the first of a set of Blockables to become ready.

4.2 Lintel

Lintel is the placeholder for a collection of otherwise unrelated components: ones that didn't belong in some other library, typically. We'll survey its contents briefly by function.

4.2.1 Statistics objects

There are three primary statistics families: the Stats, StatsUtil, and the Correlation families. The Stats family is designed to record values of single variables. A StatsUtil object allows resource utilizations (e.g., cache space utilizations, queue lengths) to be collected: things like "75% of the time, the utilization was zero". A Correlation object records the relationship between pairs of values, and is able to report their covariance and correlation coefficients.

Stats — collects mean, count, and standard deviations.
  StatsHistogram — also keeps a histogram of the number of times values fit in each of a set of bins
    StatsHistogramUniform — all bins are the same size
    StatsHistogramLog — the width of the bins increases exponentially
      StatsHistogramLogAccum — adds up the values, as well as the counts in each bin

StatsUtil — basic summary of how long a resource spends at different utilization levels
  StatsUtilHistogram — also keeps a histogram of times at each level

Correlation — keeps basic correlation information for two values
  CorrelationDensity — also keeps a 2D histogram of counts of values in an array of bins

In use, Stats objects are attached to *measurement points*: C++ pointers to a Stats object of the appropriate class. By convention, these are initialized to NULL until the associated Stats object is created and assigned. This means that testing whether statistics should be gathered is a simple, quick test:

```
if (sample_measurement_point != NULL)
    sample_measurement_point->add(value);
```

The IOitemStats object is simply a collection of these measurement-point hooks, suitable for recording all the interesting things about a collection of IOitems, such as execution times for reads, writes and all requests combined.

The Tcl code uses the value of the variable StatsLevel to determine how many Stats objects to construct, and what level of detail they should record. By default, summary statistics are accumulated in most places, and full histograms at the IOload and device level. Increasing the StatsLevel will cause more Stats objects to be created, and/or cause more histograms to be recorded.

In addition, every Stats object is put onto a Tcl list called the StatsList: at the end of each batch, the objects on this list are polled and asked to report their values. They may also be asked to *reset* themselves (zero their counters, etc)—for example, this is usually done at the end of the first, or warm-up, batch that gets the system into steady state.

4.2.2 Reporters

Stats and other simulation objects accumulate state information during the course of a simulation run; Reporters enable them to emit it. By analogy with the StatsLevel variable, there's a PrintLevel and a PrintList: the former determines how often things are printed (just once at the end of each run, or after every batch); the latter collects all the things that can be asked to report themselves.

Any object that wishes to emit results must be of type Reportable, which means that it supports the report function. This takes as argument a Reporter, through which results are emitted, one value at a time, the whole being bracketed by a begin/end call pair. The job of the Reporter is to format the output; the default one emits the values as Tcl expressions containing lists of name-value pairs.

4.2.3 Random number generators

Lintel provides a range of random-number generators, all derived from drand48(3C). The base class for the random-number generators is a distribution, which supports a draw() operation to provide a new value. The different subclasses provide different kinds of distributions: Crand always returns the same value (a constant); Urand returns uniformly-distributed numbers over its range; Erand returns ones drawn from an exponential distribution with the given mean; and Nrand provides a normal distribution with a given mean and standard deviation.

The simulator initializes all its random number generators from a single seed generator (a random number generator that emits values used as seeds). This mechanism means that all the random number generators don't start out with the same seed value, so (for example) not all the synthetic workload generators attempt to write to exactly the same block on their first request. By default, the seed generator is itself initialized with a seed of zero, so that runs are repeatable. For truly random behavior between runs, the seed generator can optionally be initialized with a seed derived from the low-order bits of the time-of-day clock.

4.2.4 Storage management

Since Pantheon spends much of its time allocating and deallocating small objects, Lintel provides some software that makes this all go much faster, by pre-allocating pools of objects of the appropriate sizes, and reusing returned objects rather than allocating new ones wherever possible. These are encapsulated in the files {Bucket, BucketT, CheckMalloc, StorageManagement}.[HC], and enabled by the conditional compilation flag STORAGE_MANAGEMENT.

Many of the Pantheon simulation objects use the convention of an isAssigned flag that indicates "this object is in use" to catch the common error of deleting an object prematurely.

4.2.5 Tcl to C++ interface

The interface between Tcl and C++ is straightforward in principle, but complicated in its details. Each time a new simulator object class is written, it is given a Tcl *interface function* with the same name as the object class. Calling this interface function from Tcl instantiates a new C++ object of the given class, gives it a Tcl name, and instantiates a set of Tcl "dot functions" that invoke C++ operations on it. Most of the interface is encapsulated by macros.

The result is that C++ objects can be constructed from Tcl quite simply. For example:

```
set new_link [Link "sample" -b 20e6 -o 2e-6];
```

creates a new Link with bandwidth 20MB/s and overhead 2µs, and gives it the shorthand name sample. Its actual name will be something like ":Link:2:sample:", and this string will also be the result of the Link Tcl function. Various functions with names of the form "$new_link.*something*" are created at the same time so that it is easy to invoke certain of the Link's member functions. For example,

the following Tcl code creates a new Stats object and attaches it to the Link's claimTime measurement point:

```
$new_link.claimTime [Stats "sample_claimTime"];
```

More details of this interface are provided in [Golding94].

In support of all this, the class Nameable allows objects to be given string names, and to record and print them when useful, and the class NameValueDatabase provides a structure used to map between Tcl string names and the C++ object addresses needed to make the interface work. There are several Tcl-related header files:

- TclExecution.H: makes a few of the Tcl interpreter functions visible to C++ code.
- TclInterfaceFunctions.H: provides the macros used for writing Tcl interface functions.
- TclInterfaceTypes.H: provides Tcl-to-C++ and back again conversion functions for all the basic types supported and used in Pantheon.
- TclInterfaceVariables.H: supports the interchange of values between C++ and Tcl variables: several things (e.g., StatsLevel) are both Tcl variables and C++ globals; this code keeps them in step, exporting values across the boundary as needed.

4.2.6 Error handling

Experience has taught us that liberal use of invariant assertions and a convenient set of error-reporting functions are a great boon to debugging a complicated simulation. This set of functions makes all that easier.

- Assert.H: provides the Assert() function to test invariants, and invoke appropriate error handling code if not needed. Unlike the traditional C assert() function, an Assert call has a level: assertions at a particular level are only checked if the value of the global AssertLevel is at least that large. This means that very thorough, expensive invariant checking code can be written and left in place, and need cost little at runtime unless it is enabled.

  This package also includes an Abort function that lets functions be *registered* to be called when something goes wrong, to allow application-specific debugging routines to be invoked before a core dump is taken.
- ErrorHandling.H: used to send warning messages to stdout.
- ErrorHandlingMemory.H: exits gracefully if new runs out of space.
- Log.H: some simple code to generate logging files. (Most of the simulator uses the LOG_EVENT and LOG_IOITEM series of macros defined in Pantheon.H and IOitem.H respectively.)

4.2.7 Queues, lists and other data structures

Lintel also provides a collection of modules to support various useful data structures. (Well, we think so, anyway.)

- Bitmap: records 0/1 values for a range of addresses in a compact form: think of it as a dense array of flags.
- HashTable: implements a simple hash table with collisions resolved by chaining.
- HeapSort: a heapsort class that uses templates.
- IntSeq: permutation and other simple integer sequence functions.
- List, ListT: doubly-linked lists of things. The ListT variant uses templates instead of void *.

- PQueue, Queue, QueueT: various types of queues.
- TimeStamp: a time value that records seconds and microseconds as separate words (this structure is used in the SRTlite traces).

4.2.8 And a few other bits and pieces

- LintelMisc.H: type definitions and #defines for simple types and values used throughout Lintel and Pantheon.
- LintelUtilities.H: miscellaneous utility functions such as templated gcd and lcd.

4.3 Workloads

Over the years, we have collected a number of I/O traces at the disk driver and SCSI bus levels. Although some of them are available only to people inside HP because of confidentiality reasons, the ones described in [Ruemmler93] are available to pretty much anybody under a non-disclosure agreement, both to allow them to explore interesting I/O system design issues, and to make it possible to replicate our own published work. Since the set of traces is constantly changing, it won't be detailed here.

We have chosen to convert all the various input trace formats we received into a common format, known as SRT (the meaning of the acronym is lost in the mists of time). The longer traces are broken up into 1-day units; they can be concatenated together, or treated as semi-independent files. For example, we sometimes synthesize a heavier load by replaying different weeks of traces for the same system in parallel.

4.3.1 *SRTlite filters and their composition*

The original SRT library required a couple of Bison parsers to understand the header information encoded in the traces, and was used to construct a simulator that corresponded to the description found there. Since we now use a different way of building simulators, we've also rewritten the trace-access library, which is now called *SRTlite*.

Consistent with our philosophy of composing bigger systems out of smaller components, the SRTlite library is itself composed of a number of different elements that can be composed together for various purposes to form a graph of components that feed one or more traces into the simulator code proper. Some examples of the possibilities include:

- merging multiple streams together into one;
- subsetting an input stream, e.g., by request size or type;
- modifying elements of the requests in a stream, such as their start addresses;
- speeding up (or slowing down) the rate at which requests appear to arrive, e.g., to partially model the effects of a faster processor or heaver load.

These functions are described in greater detail in a separate document [Ganger95].

In addition to the real traces we have available, SRTlite can read in an ASCII stream that specifies much the same things, and convert it into a stream of IOrequest objects. There's also a module that can synthesize a set of requests using a set of arrival-rate and other distributions, although this code is highly experimental in nature and there remains much room for improvement before we can adequately emulate a real-world stream [Ganger95a].

5 Running the simulator

This section describes how the Pantheon simulator is typically used, and how the Tcl files that drive it are structured. It also briefly discusses the interface to the DQS batch system, and the Oculus display interface.

5.1 Pantheon

Pantheon itself supports just a few command-line switches:

```
Usage:
        Pantheon [-h] [-Z debug_level] [-P print_level] [-B block_level] \
        [-X val,var,expr] [-f file] [-e expression] ...
Where:
        -h              issue this message
        -Z debug_level  set debugging verbosity [DebugLevel]
        -P print_level  set printout  verbosity [PrintLevel]
        -B block_level  set level of blocking debugging operations [BlockLevel]
        -X val,var,expr delay evaluation of expr until var reaches val
        -f file         file of Tcl commands to obey
        -e expression   Tcl expression to obey
```

Instead, most of the configuration work is done by the –e and –f flags. The first of these is typically used to set (or override) run-specific Tcl variables; the second is then used to read in the Tcl files controlling the simulation construction.

The typical sequence of files that is read in is as follows:

- init.tcl: pre-loads any necessary libraries (currently only used by Oculus);
- traces.tcl: determines which trace files to read— it contains functions that map various shorthand specifications (e.g., 1day, 1week) into the appropriate trace files for a particular host; it is also associated with a set of files called *hostname*.tcl that describe the configuration of the host systems from which the traces were taken;[1]
- *backend*.tcl: there's one file for each major experimental setup (e.g., host.tcl builds a simulator that emulates the original configuration of the traced host); a common idiom here is an *xxx*Defaults.tcl file that sets default values for various parameters for an *xxx* system under test, unless they have already been defined by a –e expression argument;[2]
- experiment.tcl: constructs the Experiment, SRTlite filter graph, and the associated IOloads and IOloadGenerators;
- simulate.tcl: runs the simulation proper, executing a batch of work, emitting results, and then continuing on if appropriate.

5.2 runPantheon

A front-end script, runPantheon, provides the most usual way to invoke Pantheon. It has a multitude (somewhere beyond a wealth?) of options, which are gradually being extended as we add new features to the simulator. To find out the current set, invoke:

```
runPantheon '-?'
```

This will provoke something like this:

1. This function really ought to be moved into the per-host description files, but we haven't had a chance to do so yet.
2. Unfortunately, other examples of this genre (e.g., Array.tcl, IceCube.tcl) are not in the set that we can export outside HP.

```
Usage: runPantheon [-d|D] [-Q] [-q queue] [-C|c]  [-V lvl] [-K time] \
        [-P printlevel] [-Z debuglevel] [-z IOitemId,debuglevel] \
        [-X value,variable,expression] [-B blockinglevel] \
        -h host -b backend -t tracename [-E experiment] [-s simulate] \
        [-T “select”] [-p [-]n,n...] [-I <input command>]\
        [-f tracefile] [-F tracefamily] [-S simulator] [-o out-dir]\
        [-e “tclvar value”] [-l “tclvar value”] [-0 file] ... [-4 file] \
        [-m “merge specification file”] [-M “merge building file”] \
        [-nobatch]
```

The following are the minimum that's needed to run a simulation:

-h *host*	host system (hplajw, snake, cello, ATT, netware-*)
-b *backend*	back-end system (host, IceCube)
-t *tracename*	duration/name of trace (1day, 1week, shortweek, all)

The following flags are passed through to Pantheon itself:

-P *level*	set print level
-Z *level*	set debug level
-z *Id,level*	set debug level starting at IOitem Id
-X *val,var,expr*	obey Tcl “expr” when delay-variable “var” gets to “val”
-B *level*	set blocking debug level
-e “*tclvar value*”	set Tcl variable before running Tcl scripts
-l “*tclvar value*”	lappend value onto Tcl variable before Tcl scripts

The following options control which trace inputs are used:

-f *tracefile*	full name of trace .srt file
-F *tracefamily*	name of tracefamily (default is “1992-complete-disk”)
-T “*filter*”	filter records from the trace
-p *[-]n,n…*	select or eliminate disk n from the trace
-m *file*	file that specifies mergegroups
-M *file*	file that remaps and filters devices
-I “*command*”	a command whose output to pipe into Pantheon

The following are concerned with foreground/background job submission, and whether the simulator is run, or debugging information emitted instead:

-d	emit generated Pantheon arguments to stdout
-D	emit generated command to stdout
-Q	build a dqs job and submit it (-D overrides submit)
-q *queue*	queue to submit to (no-op if no -Q)
-C	keep any generated core file (default for non-DQS)
-c	delete any generated core file (default for DQS)
-K *time*	kill the Pantheon run after “time” seconds
-V *level*	make runPantheon debug itself
-O *level*	invokes userinterface Oculus with OculusLevel
-o *out-dir*	prefix for output directory name (default = “output”)
-nobatch	do not generate intermediate batch files
-S *simulator*	default is Pantheon, “” means no simulator
-E *experiment*	name of experiment to run (default is “experiment”)
-s *simulate*	name of simulation-control file (default is “simulate”)
-0 *file* … -4 *file*	Normal sequence of tcl control files is host, back-end, experiment, simulate these flags insert auxiliary Tcl files in positions shown: *file0* host.tcl *file1* back-end.tcl *file2* experiment.tcl *file3* simulate.tcl *file4*

Warning: runPantheon does not conform to the UNIX getopt conventions: instead, all flags and their values must be passed in as separate arguments (i.e. –V4 is not allowed: you must write –V 4)

5.3 The node graph

The topology of the simulation is described is via a declarative structure called the *node graph*. This captures the relationship of hosts and the devices they are connected to, as well as the routes by which this is done. This graph is used to determine what simulation objects will be constructed. Changes in what is built are accomplished by modifying the node graph before the construction step. For example, to model the HP AutoRAID, we removed all the physical disks from the node graph and replaced them with a single AutoRAID device. More information on the how to do this is available in a separate document [Abram95].

5.4 DQS support

As mentioned above, DQS support is included in runPantheon. The requirements are fairly straightforward: header and trailer scripts are added to the set of commands generated to execute Pantheon, and the result spooled off to DQS's job-submission program.

The rest of the support is provided in the Adze tools, which are used to bring back results files from where they end up—typically in the local file systems of the compute processors. Once the results have been retrieved, they can be analyzed with Adze or Tongs.

Adapting runPantheon and Adze to use another kind of batch submission facility should be straightforward.

5.5 Oculus

Oculus allows the state of the simulator internals to be monitored while the simulator is running. It does this by providing a special kind of Reporter function, and triggering a call of report() much more often than at the end of every batch. The output from these is sent across a pipe to the Oculus program proper, which is a Tcl script that can create windows for each of the statistics objects being described by the reports, including things like moving strip-charts of the values being emitted.

Thanks to the way it is constructed, Oculus imposes absolutely no overheads on the simulator when it is not in use. To enable it, supply the –O flag to runPantheon.

> *The Oculus program is supplied on an as-is basis: thanks to some internal incompatibilities in the versions of Tcl, Tk, and various other Tcl libraries that it uses, it has temporarily stopped working. Sorry.*

6 Conclusions

This paper has provided a birds-eye view of the contents and workings of the Pantheon simulator and some of its support tools. We've had fun building them, and even more fun using them; we hope you'll get as much out of pantheon as we have.

Disclaimers

Pantheon is offered to other research groups on an "as-is" basis. We regret that we are unable to provide any support for it. However, we would like to receive notices of bugs that you find and fix, so that we can propagate these to our own version as well as others who may be using the tool, and we encourage you to make any useful modules that you construct for Pantheon available to us for the same reason.

Regrettably, commercial considerations preclude us from giving away everything that we've develope for Pantheon to research teams outside HP. In particular, our disk array models and the ones for the HP AutoRAID fall into this category. Sorry.

We solicit your inputs and suggestions as to how we could make the Pantheon suite a more valuable tool—although we can't promise to make any changes, we will listen to any input you provide.

Acknowledgments

Many people have helped build the Pantheon environment to the state it is in now. They include past and present members of the Storage Systems Program (Liz Borowsky, Richard Golding, Arif Merchant, Mirjana Spasojevic, Carl Staelin, Tim Sullivan, John Wilkes), some of our other HP Labs colleagues (David Jacobson, Milon Mackey), and a raft of summer students and visitors (Uwe Aicheler, Peter Bosch, Pei Cao, Greg Ganger, Swee Boon Lim, Stefan Savage, Shivakumar Venkataraman, Janet Weiner, Bruce Worthington, Rebecca Wright). We thank them all.

References

[Abram95] Michelle D. Abram. *Arbitrary topology specification for TickerTAIP*. Technical Report HPL–SSP–95–4. Storage Systems Program, Hewlett-Packard Laboratories, Palo Alto, CA, 30th August 1994, issued 30th August 1995.

[Aicheler96] Uwe Aicheler. *Oculus: a visual user interface for the Pantheon storage system simulator*. Technical Report HPL–SSP–96–1. Storage Systems Program, Hewlett-Packard Laboratories, Palo Alto, CA, 8 January 1995.

[ATT89] AT&T. *Unix System V AT&T C++ language system release 2.0 selected readings*, Select Code 307-144, 1989.

[Cao94b] Pei Cao, Swee Boon Lim, Shivakumar Venkataraman, and John Wilkes. The TickerTAIP parallel RAID architecture. *ACM Transactions on Computer Systems*, **12**(3):236–69, August 1994.

[Ganger95] Gregory R. Ganger and John Wilkes. *The SRTlite library: trace access, generation and manipulation*. Technical Report HPL–SSP–95–15. Storage Systems Program, Hewlett-Packard Laboratories, Palo Alto, CA, 1 September 1994, issued 29 December 1995.

[Ganger95a] Gregory R. Ganger. Generating representative synthetic workloads: an unsolved problem. *Proceedings of CMG'95* (Nashville, TN), December 1995.

[Golding94] Richard Golding, Carl Staelin, Tim Sullivan, and John Wilkes. "Tcl cures 98.3%of all known simulation configuration problems" claims astonished researcher! *Proceedings of Tcl/Tk Workshop, New Orleans, LA*. Computerized Processes Unlimited, Incorporated, June 1994. Also published as Technical report HPL–CCD–94–11, Concurrent Computing Department, Hewlett-Packard Laboratories, Palo Alto, CA.

[Golding95] Richard Golding, Peter Bosch, Carl Staelin, Tim Sullivan, and John Wilkes. Idleness is not sloth. *Proceedings of Winter USENIX 1995 Technical Conference* (New Orleans, LA), pages 201–12. Usenix Association, Berkeley, CA, 16–20 January 1995.

[Golding95a] Richard Golding. *Data extraction and display using Tongs*. Technical Report HPL–SSP–95–16. Storage Systems Program, Hewlett-Packard Laboratories, Palo Alto, CA, 29 December 1995.

[Golding95b] Richard Golding. *The Raphael threads library.* Technical Report HPL–SSP–95–17. Storage Systems Program, Hewlett-Packard Laboratories, Palo Alto, CA, 29 December 1995.

[Green93] Thomas P. Green and Jeff Snyder. *DQS, a Distributed Queueing System*, Version 2.1. Supercomputer Computations Research Institute at Florida State University, 13 March 1993.

[Jacobson91] David M. Jacobson and John Wilkes. *Disk scheduling algorithms based on rotational position*. Technical report HPL–CSP–91–7. Hewlett-Packard Laboratories, 24 February 1991, revised 1 March 1991.

[Keppel93b] David Keppel. *Tools and techniques for building fast portable threads packages*. Technical report 93–05–06. Department of Computer Science and Engineering, University of Washington, 1993.

[Ousterhout94] John K. Ousterhout. *Tcl and the Tk toolkit*, Professional Computing series. Addison-Wesley, Reading, Mass. and London, April 1994.

[Pawlikowski90] Krzysztof Pawlikowski. Steady-state simulation of queueing processes: a survey of problems and solutions. *Computing Surveys*, **22**(2):123–70, June 1990.

[Ruemmler93] Chris Ruemmler and John Wilkes. UNIX disk access patterns. *Proceedings of Winter 1993 USENIX* (San Diego, CA), pages 405–20, 25–29 January 1993.

[Savage96] Stefan Savage and John Wilkes. AFRAID—A Frequently Redundant Array of Independent Disks. *Proceedings of Winter USENIX* (San Diego, CA), January 1996.

[Seltzer90b] Margo Seltzer, Peter Chen, and John Ousterhout. Disk scheduling revisited. *Proceedings of Winter 1990 USENIX Conference* (Washington, D.C.), pages 313–23, 22–26 January 1990.

[Soepenberg95] Gerben Soepenberg. *A storage system I/O replayer.* Technical Report HPL–SSP–95–5. Storage Systems Program, Hewlett-Packard Laboratories, Palo Alto, CA, August 1995.

[Wilkes96] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID hierarchical storage system. *ACM Transactions on Computer Systems*, **14**(1), February 1996.