ML-RSIM Reference Manual

Lambert Schaelicke

Department of Computer Science and Engineering

University of Notre Dame

Mike Parker

School of Computing

University of Utah

University of Notre Dame
Department of Computer Science and Engineering
Technical Report 02-10

November/02

CONTENTS

Part I: Simulator Overview

1 Introduction 1
2 System Architecture 2
  2.1 System Architecture Configuration 3
  2.2 Address Map 3
  2.3 Simulator Page Table 4
3 Parameter Setup 5
4 Simulation Input/Output Files 6
5 Multiprocessor Simulation 7
6 Kernel Image 8
7 File Descriptor Management 9
8 User Statistics 10
  8.1 User Statistics Management 10
  8.2 User Statistics Structures 10

Part II: Processor Model

1 Overview 11
2 Fundamental Data Structures 13
  2.1 Dynamic Instructions 13
  2.2 Activelist 13
  2.3 Stall Queues 13
  2.4 Heaps 14
  2.5 Processor State 14
3 Parameters 15
4 Register File 17
  4.1 Register Definition 19
  4.2 Register Access 20
5 Little Endian Host Support 21
6 Instruction Fetch Stage 22
7 Instruction Decode Stage 24
  7.1 Instruction Decode 24
  7.2 Dependency Checking 24

7.3 Context Synchronization . . . 25
8 Instruction Issue and Completion . . . 26
8.1 Issue . . . 26
8.2 Completion. . . . 26
9 Instruction Graduation Stage . . . 27
10 Memory Instruction . . . 28
10.1 Memory Instruction Issue . . . 28
10.2 Memory Instruction Completion . . . 28
10.3 Memory Disambiguation . . . 29
10.4 Address Generation . . . 30
10.5 Uncached Load/Store Instructions . . . 30
10.6 Memory Barrier & Uncached Buffer . . . 30
11 Exception & Interrupt Handling . . . 31
11.1 Trap Table . . . 32
11.2 Exception Detection . . . 32
11.3 Exception Handling . . . 32
11.4 System Call Traps . . . 33
11.5 External Interrupts . . . 33
11.6 Statistics . . . 34
12 System Calls and Simulator Traps . . . 35
12.1 System Call Interface . . . 35
12.2 Simulator Trap Handling . . . 35
12.3 Simulator OS Bootstrap Information . . . 35
12.4 File I/O Simulator Traps . . . 36
12.5 Socket Simulator Traps . . . 38
12.6 Statistics Simulator Traps . . . 38
12.7 Miscellaneous Simulator Traps . . . 39
13 Instruction & Data TLB . . . 40
13.1 TLB Configuration. . . . 40
13.2 TLB Entry Format . . . 41
13.3 TLB WriteEntry . . . 42
13.4 TLB LookUp . . . 42
13.5 TLB Probe and Flush . . . 43

Part III: Memory Hierarchy

1 Overview . . . 44
1.1 Request Structure . . . 45

1.2 Node Initialization and Statistics 47

2 Caches 48

2.1 Cache Module Structure 48

2.2 Cache Initialization 49

2.3 Cache Statistics 51

2.4 L1 Instruction Cache 51

2.5 L1 Data Cache 53

2.6 L2 Cache 53

3 System Control Module 56

3.1 Address Map 56

3.2 System Module Initialization 57

3.3 Handling CPU Requests 57

3.4 Handling External Requests 58

4 Uncached Buffer 59

4.1 Initialization 59

4.2 Issuing Uncached Memory Operations 60

4.3 Uncached Buffer Structure 60

4.4 Handling Incoming Requests 61

4.5 Sending Requests to the System Bus 62

4.6 Statistics 62

5 System Bus 63

5.1 Bus Configuration 63

5.2 Bus Transactions 64

5.3 Trace File 65

5.4 Bus Operation 65

5.5 Bus Statistics 67

6 Main Memory Controller 68

6.1 Memory Controller Structure 69

6.2 Request Handling 69

6.3 Coherency 70

6.4 Statistics 70

7 DRAM Model 71

7.1 DRAM Controller Structure 73

7.2 DRAM Operation 74

Part IV: I/O Subsystem

1 I/O Devices 76

1.1 Initialization . . . 77
1.2 Handling Incoming Bus Transactions . . . 77
1.3 Outgoing Bus Transaction Handling . . . 78
1.4 Data Reply Handling . . . 79
1.5 Request Completion for I/O Devices . . . 79
2 Realtime Clock . . . 80
2.1 Operation . . . 80
2.2 Initialization . . . 81
2.3 Clock Event . . . 81
2.4 Reading and Writing Registers . . . 81
2.5 Statistics . . . 82
3 PCI Subsystem . . . 83
3.1 PCI Bridge Initialization . . . 83
3.2 PCI Device Registration . . . 84
3.3 PCI Bridge Operation . . . 84
4 SCSI Subsystem . . . 86
4.1 Data Structures . . . 87
4.2 SCSI Requests . . . 88
4.3 SCSI Bus Model . . . 89
4.4 SCSI Devices . . . 90
5 SCSI Host Adapter . . . 91
5.1 Generic SCSI Adapter Module . . . 91
5.2 Adaptec AIC7770 SCSI Controller Model . . . 93
6 SCSI Disk . . . 101
6.1 Configuration . . . 101
6.2 Request Handling . . . 102
6.3 On-board Cache . . . 105
6.4 Disk Mechanics . . . 106
6.5 Persistent Storage . . . 107
7 Platform Specific Details . . . 108
7.1 Memory References . . . 108
7.2 Simulator Traps . . . 108
7.3 Floating Point Arithmetic and Exceptions . . . 108

Part V: The Lamix Kernel

1 Overview . . . 110
1.1 Compiling and Linking ML-RSIM Applications . . . 110

1.2 Memory Layout 111
1.3 Page Table Organization 113
1.4 Process Structures 114
2 Startup and Initialization. 117
2.1 Init Process Setup. 118
2.2 Idle Processes. 118
2.3 SMP Startup (not enabled yet). 118
2.4 Kernel Parameters 119
2.5 Kernel Memory Allocation 121
2.6 Kernel Patching 121
3 Processes. 122
3.1 Process Creation. 122
3.2 Process Termination. 122
3.3 Kernel Threads. 124
3.4 Process Statistics 124
3.5 Program Execution. 124
3.6 Context Switch. 125
3.7 Synchronization 126
4 Exception Handlers. 128
4.1 TLB Miss Handler 128
4.2 Generic Trap Handling 129
4.3 Memory Protection Fault Handler 129
4.4 Bus Error 130
4.5 Miscellaneous Error Traps. 131
4.6 System Trap Handler 132
4.7 Interrupt Handler 133
5 System Calls 134
5.1 Fork System Call 134
5.2 Indirect System Call. 134
5.3 Illegal System Call. 134
5.4 Brk System Call 135
5.5 Process Attribute System Calls 135
5.6 System Attribute System Calls 136
5.7 User-Context System Calls 136
6 Signals. 139
6.1 Initialization 139
6.2 Signal System Calls 139
6.3 Signal Transmission. 141

6.4 Signal Delivery 141
7 Timers 143
7.1 Interval Timers 143
7.2 Realtime Timer Event Handler 144
8 Shared Memory 145
8.1 Segment Descriptors 145
8.2 System Calls 145
9 User Statistics 147
10 File Systems 148
10.1 Virtual Filesystem Layer 148
10.2 Buffer Management 149
10.3 Filesystem Types 150
10.4 Host File System 151
10.5 BSD Fast Filesystem 154
10.6 BSD Log-structured Filesystem 154
11 Networking 155
11.1 Networking System Calls 155
11.2 Socket Routines 155
11.3 Protocol Routines 156
11.4 Unix Protocol 156
11.5 Internet Protocol 157
12 I/O Subsystem and Device Drivers 159
12.1 Realtime Clock 160
12.2 Memory Device 161
12.3 PCI Driver 161
12.4 Adaptec SCSI Adapter 162
12.5 SCSI Disk 165
12.6 RAIDFrame 166
12.7 Concatenated Device Driver CCD 167
13 Autoconfiguration 168
13.1 Autoconfiguration Utility Routines 168
13.2 General Device Configuration Routines 169
13.3 Mainbus Configuration 169
13.4 CPU Configuration 169
13.5 PCI Bus Configuration 169
13.6 SCSI Bus Configuration 171
13.7 Pseudo-device Configuration 171

Part VI: Libraries

1 LibSocket 173
2 LibPosix4 174
3 LibUserStat 175

Part VII: System Administration Tools

1 Disklabel 177
2 NewFS. 179
3 NewLFS 180
4 LFS_Cleanerd 181
5 FSCK 182
6 RaidCtl 183
7 CCDConfig 185

Appendix A: Directory Structure 187

Appendix B: Summary of Parameters 190

Part I: Simulator Overview

1 Introduction

The ML-RSIM simulation system is based on an event driven simulation library YACSIM. This library controls the scheduling of event handling routines and maintains the notion of a global time. Only a subset of YACSIMs features are used in the simulator.

An event is an action that is scheduled to occur at a particular time. Each event consists of a function body with associated arguments, a state variable and an invocation time. When the simulation time is equal to the invocation time, the function associated with the event is invoked with the arguments that where specified when the event was scheduled. The function can reschedule the event for the future, delete the event, inquire about the event state, modify the state and schedule other events.

Before an event can be scheduled, an event data structure must be allocated and initialized by calling *NewEvent*. This routine takes the event name, the event body and two flags as arguments and returns a pointer to the event structure. This pointer can be used to obtain (*EventGetState*) or modify (*EventSetState*) the event state. The event state is essentially an implicit argument to the event routine that can be used to maintain state across event invocations. The macro *schedule_event* can be used to schedule a given event at a specific time in the future. Care must be taken that an event is not scheduled twice, this will lead to cycles in the event list and hang the simulator.

The simulator uses three different scheduling mechanisms. The processor model is assumed to be busy almost every cycle, hence it is scheduled every cycle. The routines that correspond to individual pipeline stages check if any work needs to be done. Each processor also maintains a DELAY variable that indicates if the processor is idle for a known number of cycles.

The caches are only simulated when necessary. Each cache maintains two flags that indicate if requests are waiting in any input queue, or if requests are being processed in the cache pipelines. This check is performed every cycle, but the corresponding simulation routines are only called when necessary.

Other system modules such as the bus, memory controller and I/O devices are completely event driven. When a request is issued onto the bus by a cache module, the bus event is scheduled after a fixed delay. Similarly, the memory controller and I/O device events are only scheduled when a request for this module is processed by the bus.

2 System Architecture

The simulator is able to simulate multiple independent nodes, each of which may contain more than one processor. The figure below shows the high-level architecture, along with the relevant configuration parameters. Note that currently, ML-RSIM does not provide a model of a network connecting individual nodes.

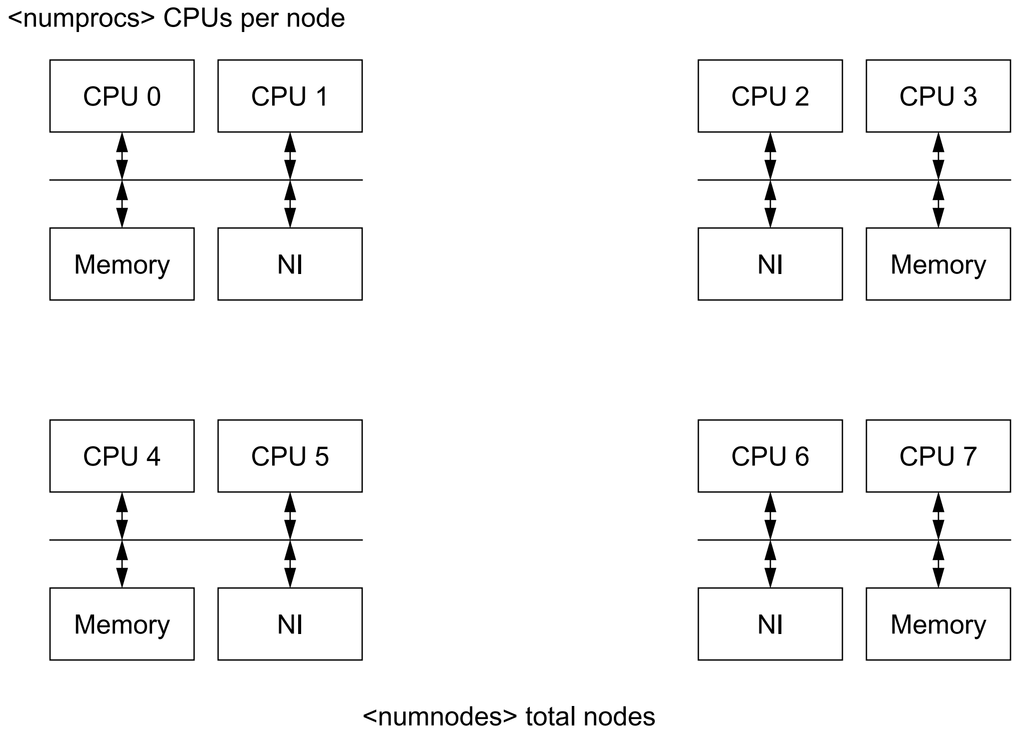

Figure 1: ML-RSIM System Architecture

In the above configuration, each node contains 2 processors. Processors are numbered consecutively across nodes. Within each node, modules are numbered starting at 0 (first processor), with the memory controller being last. Requests are identified by a pair of node and module IDs, which are also used as index into the various pointer lists for bus modules. For instance, a processors global ID, which is also the index in the processor structure list, is computed as: *proc_id + num_nodes * node_id*.

Each module has one port to the system bus. Requests from the processor are demultiplexed by the cache module based on the memory attributes and sent either to the L1 cache or uncached buffer. The L2 cache implements the system interface, which multiplexes requests from the cache and the uncached buffer on to the bus. The memory controller (in its function as node coordinator), receives coherence responses from all coherent modules (all CPUs plus coherent I/Os) and sends the coherence state back to the original requestor via a dedicated port. This models a node architecture with variable-latency coherency reporting on dedicated buses, where the coherency status is returned with the data.

Not shown in the figure are the per-node simulator page table and address map. The address map contains the list of address ranges that modules on the bus respond to, it is used to map addresses to target modules when a request is created. The simulator page table maps simulated addresses to host addresses, it should not be confused with the operating system page table.

2.1 System Architecture Configuration

files: Caches/system.c
Caches/system.h

The total number of nodes and the number of processors per CPU can be defined by the following configuration parameters:

Parameter	Description	Default
numnodes	number of nodes in system	1
numprocs	number of processors per node	1
numscsi	number of SCSI bus adapters per node	1
numdisk	number of disks per SCSI bus	1

Table 1: System Configuration Parameters

The function *SystemInit* creates the necessary CPUs, caches, system buses, memory controllers and I/O devices. Along with each CPU it instantiates a first and second level cache with write buffer and an uncached buffer. The number of most I/O devices is also configurable, with the exception of the real-time clock which is always present. The number of SCSI controllers or disks may be set to zero, in which case no such device is instantiated.

2.2 Address Map

files: IO/addr_map.c, Caches/system.c, Caches/cache_cpu.c
IO/addr_map.h

The address map is a per-node list that maps address ranges to bus modules. It is used by any module that issues requests on the bus to look up which module responds to a given address. For instance, the function *DCache_recv_addr* performs a lookup when the processor sends a request to the cache or uncached buffer. Bus modules are expected to register at least one address range during their initialization.

Each node has its own address map. The *AddrMap_init* function, called in *System_init*, allocates a list of pointers to these maps and then creates an address map for each node. Maps are managed as dynamic arrays with an initial size of 32 entries. An entry contains lower and upper address pointers and the module identifier. Address regions start at *low* and end at *high*-1, the module number identifies the module within the node.

Bus modules must register their address regions during initialization by calling *AddrMap_insert* with node-number, lower and upper address and module ID as arguments. This function checks if the address region is empty, or if it overlaps with an existing region. If this is not the case, it inserts the entry into the map, possibly extending the array. Note that there are independent address maps for each node, therefore modules must register their address range with each node.

Before sending a request to the bus, bus masters can map the destination address to a target module by calling *AddrMap_lookup*, with the node-ID and physical address as arguments. The function returns the local module ID upon success, or prints an error message and exits if no matching entry was found.

2.3 Simulator Page Table

files: Processor/pagetable.cc, Processor/procstate.cc
Processor/pagetable.h

Each node maintains a simulator page table that is used to translated simulated physical addresses to host addresses. The page table is shared by all modules within a node. For backward compatibility, processors maintain a pointer to the page table in their internal data structure.

The function *PageTable_init* allocates an array of page table pointers and then creates a page table object for each node. The processor objects, as well as all other bus modules, will later use this array to locate the page table for their respective node.

Page table entries are inserted either by calling the class method *insert* from C++ code (as is done by the processor), or by calling the standard C wrapper function *PageTable_insert* or *PageTable_insert_alloc*. The latter function also allocates a page of host memory before inserting the mapping, and installs the new entry only if the physical address has not been mapped previously. During program execution, the Lamix operating system will allocate host memory and install the mappings for regular memory segments whenever necessary (for instance to grow the stack or heap). I/O devices should use *PageTable_insert* to allocate host memory as backing for control registers or buffers. This allows read and write accesses to I/O addresses to be treated like regular memory accesses by the simulator, although these locations might be implemented as registers, FIFOs or SRAM.

Alternatively, a module can call the routine *PageTable_insert* and provide its own host memory. This is useful for devices that want to map complex structures representing control registers into the physical address space.

A set of functions is provided for reading and writing host memory as integers, bytes, floating point values or bits. These functions can be used by I/O devices to independently access control registers or buffers that are backed by host memory.

3 Parameter Setup

files: Processor/mainsim.cc

After the simulation library has set up its internal data structures, it calls the routine *UserMain*. This routine reads the command line parameters, sets up the input and output files and reads the parameter file. It then initializes all modules in the system, based on various configuration parameters and calls the main simulation routine *RSIM_EVENT*. This routine performs all operations associated with simulating one cycle for the caches and processor and reschedules itself at the end of the cycle.

Parametrization of the simulator is done in two stages. Command line options are used to specify the simulation executable, the parameter file and various input/output features. The parameter file specifies all system parameters, such as functional units, cache size and bus frequency. The following table lists the command line options that are currently supported:

Parameter	Description	Default
-D <dirname>	input/output directory	current directory
-S <subject>	prefix for input/output files	rsim
-z <configfile>	configuration file path and name	rsim_params
-e <username>	send mail to user when simulation completes	NULL (don’t send mail)
-c <cycles>	maximum number of cycles to simulate	+INF
-d b	dump flag; enable bus trace file	NULL (no trace file)
-m <procs>	parallel simulation on up to N processors	off
-n	‘nice’ process - lower simulator priority	off
-t <cycles>	enable debug output after N cycles (must be compiled with COREFILE set)	0
-F <file>	simulation executable path and name	-
-u	enable fast functional units (1 cycle latency)	off
-X	static scheduling	off

Table 2: Command Line Parameters

After reading and processing the command line options, the parameter file is opened. Reading parameters from this file is distributed among the various modules. Each entry in the configuration file consists of a name-value pair.

If multiprocessor simulation is enabled and more than one node are simulated, the routine *DoMP* determines a suitable configuration, sets up shared synchronization data and forks the other simulation processes.

4 Simulation Input/Output Files

files: Processor/mainsim.cc

A simulation run normally involves five different input/output files. The simulator writes trace information, warnings and error messages in a ‘log’ file. The file is created during simulator startup, either in the current directory or in the output directory if one is specified with the ‘-D’ option. The filename is a concatenation of the subject as specified with the ‘-S’ option or the executable file name if no subject is specified, and the extension ‘.log’. Statistics output is written to a ‘stat’ file, which is created in the same directory as the log file and also uses either the subject string specified by the ‘-S’ option or the executable filename as base name.

During the boot process, the simulation operating system opens the standard set of input and output files. For this purpose, the simulator provides a trap that communicates the user-specified filename for stdin, stdout and stderr to the OS. These files will also be created in the output directory if specified, their names start with the subject if one is provided and the files have the extensions ‘.stdin’, ‘.stdout’ and ‘.stderr’.

If more than one node are simulated, either in uniprocessor or multiprocessor simulations, each node uses a private copy of the above mentioned files. In this case, each filename (except the input filename) is appended with a number indicating which node it belongs to.

5 Multiprocessor Simulation

files: Processor/multiprocessor.cc, Processor/lock.s
Processor/multiprocessor.h

If the command line flag '-m' is specified with a processor count greater than 1, multiple nodes are simulated and the host system is a shared memory multiprocessor, the simulator runs as a parallel application. Each process simulates a subset of the total number of nodes. Although each process instantiates the entire machine model for all nodes, it simulates only a subset of the simulators assigned to it. In addition, self-activating device models such as the real-time clock schedule events only for the particular subset of the nodes.

The '-m' command line flag specifies the maximum number of processes for the simulation, this may be reduced if the host system has less CPUs, the number of simulated nodes is less than the number of host CPUs, or when a more balanced load can be achieved. For instance, when simulating 5 nodes on a 4 processor system, it makes more sense to assign two nodes to two processes each and the remaining node to a third processor, than to assign one node to three processes each and the remaining 2 to the fourth process, as in either case simulation speed is dominated by the process that simulates 2 nodes, but the former case minimizes the number of CPUs utilized on the host.

Parallel instances of a simulation stay synchronized with respect to the simulated clock frequency by means of a shared memory barrier. When simulating with more than one process, the setup routine allocates a shared memory region and schedules a barrier event to occur after a fixed number of cycles. The barrier is a shared memory implementation using a counter and a broadcast flag which is toggled when the last process reaches the barrier. The barrier event reschedules itself until all simulated processors have exited. A global processor count variable protected by a lock is used to keep track of the number of active CPUs. Note that synchronization happens only at fixed intervals, in between the simulation processes proceed asynchronously.

6 Kernel Image

files: Processor/mainsim.cc, Processor/startup.cc, Processor/config.cc Processor/mainsim.h

After initializing the system, the simulator loads the kernel image into the simulated memory. The kernel filename and path can be specified in the configuration file.:

Parameter	Description
kernel	path to kernel file

Table 3: Kernel File Parameter

If a kernel file is not specified in the configuration file, the simulator attempts to locate it by removing the last three components from the simulator path and replacing them with *lamix/lamix*, assuming that the simulator executable is located in *bin/$ARCH/<executable>* and the kernel file is located in *lamix/lamix*. The simulator executable can be found either through the first command line parameter if an absolute path is used, or by scanning the search path.

The kernel executable must be in 32 bit ELF format. The simulator first reads the ELF header to determine the entry point and then scans the section headers for loadable ELF sections. These sections contain either text, read-only data (such as case-statement jump tables), initialized data or uninitialized data. The startup routine also keeps track of the start-addresses and sizes of the text and data segment so that these can be communicated to the kernel through the *GetSegmentInfo* simulator trap.

The startup routine also copies the application name and command line parameters onto the kernel stack and sets the stack pointer appropriately. These parameters are passed to the Lamix operating system which uses them to start the application program and pass it the required parameters.

7 File Descriptor Management

files: Processor/filedesc.cc
Processor/filedesc.h

To minimize the number of open files, the simulator implements its own file descriptor management. This allows the simulated kernel to keep a large number of host files open without actually using any kernel resources. Only files that the simulator currently operates on are open.

The simulator maintains a table that maps file descriptors to absolute filenames. Each entry contains the absolute name of the file or directory, the current file offsets and a set of file flags. Descriptors are allocated when simulated software executes the *sim_open* or *sim_creat* trap. The *FD_allocate* routine determines the absolute filename and opens the file to check permissions. It then allocates an entry while growing the file descriptor table if necessary, closes the file and returns the index of the new entry as a file descriptor to the simulated software.

File descriptors are removed when the simulated software executes a *sim_close* trap. The *FD_remove* routine marks the entry as free. To speed up the search for a free file descriptor, the routine also sets a pointer to the lowest free file descriptor.

When simulated software executes traps that operate on file descriptors, the trap handler calls the *FD_open* routine with the file descriptor as argument. This routine uses the name recorded in the file descriptor entry to open the file with the flags specified in the descriptor. It then performs a seek to the current offset and returns the real file descriptor. This file descriptor can be used to perform normal reads and writes and any other operations on file descriptors. The trap handler is responsible for updating the current file offset by calling *FD_lseek*, and for closing the file descriptor when it is no longer needed.

8 User Statistics

files: Processor/userstat.cc
Processor/userstat.h

The user statistics feature allows software to control and trigger statistics collection inside the simulator. Software creates a statistics gathering structure through a simulator trap with a unique name and the statistics type as arguments. The simulator either creates a new structure of the specified type, or returns the handle of an existing structure with the same name. Subsequently, software triggers statistics sampling by means of another simulator trap. The semantics of the sample depend on the particular type of the statistics structure.

8.1 User Statistics Management

User statistics are managed in a per-node array of pointers to statistics objects. The routine *UserStats_init* is called during system initialization, it allocates and initializes an array of pointers to the per-node structures. The routines *UserStats_report* and *UserStats_clear_all* are called when the simulator prints out statistics for the entire system, and when all statistics variables are reset, respectively. Each node maintains a private set of user statistics, managed by the *user_stats* class. This class keeps an array of pointers to the individual statistics objects, creates and initializes new objects, forwards sampling events and prints or resets the statistics objects.

New statistics objects are created by calling *user_stats::alloc* with the statistics type and a unique name as arguments. The routine searches through the list of existing objects to determine if the name already exists. If a matching name is found, its index is returned, otherwise the array is grown and a new object of the specified type is created and initialized. Print and reset events are simply forwarded to all objects in the list. The *user_stats::sample* routine takes the statistics object index and a type-specific integer as arguments and calls the *sample* routine for that particular object.

8.2 User Statistics Structures

Individual user statistics are managed by various classes that are derived from a simple base class. The base class *user_stat* implements the name of the class, provides a constructor that stores the name and a name retrieval method. Derived classes should make sure to call the base class constructor in order to initialize the object name. The *sample*, *print* and *reset* routines implement type-specific behavior. Currently, the following statistics types are supported.

- interval: treat pairs of samples as intervals and record total duration, minimum, maximum and average interval as well as number of intervals and print them in processor cycles and wall time. The user-specified value is not used.
- point: records maximum, minimum and average of all samples, as well as number of samples
- trace: print trace information, including statistics name specified during allocation and current cycle, to logfile at every sample invocation

Part II: Processor Model

1 Overview

The processor model simulates a dynamically scheduled superscalar CPU. A configurable number of instructions is fetched, decoded, issued and graduated per cycle. Instructions are issued as soon as the operands are available and the functional unit is idle. The original instruction order is maintained in an active list, which combines the concepts of an issue queue and reorder buffer. In addition, memory instructions are maintained in a separate queue which is responsible for memory load/store conflict detection.

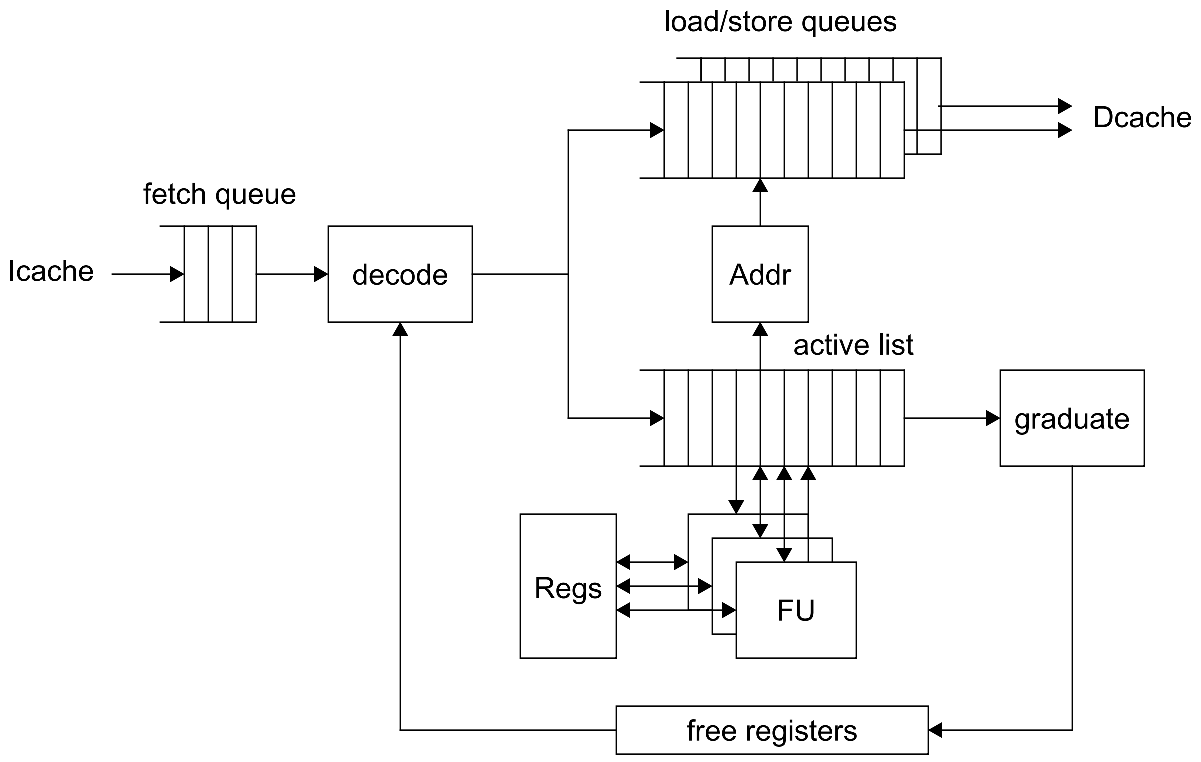

Figure 2: Processor Microarchitecture

The processor is simulated every cycle. The main simulation routine *RSIM_EVENT* first simulates cache events that produce data, then calls the processor simulation routines and then simulates cache events that consume data from the processor or other modules.

The processor simulation essentially calls routines that correspond to a CPUs pipeline stages in reverse order. In order to simulate multiple cycle stalls, each processor maintains a *DELAY* variable that is decremented every cycle. In normal operation, this variable is set to 1 at the end of a cycle. In case of an exception that needs to flush the processor pipeline, the variable is set to the number of cycles that the flush operation would take, while the actual flushing happens instantaneously.

During each cycle, the processor simulator performs the following operations:

- if an exception is pending, check if it can be taken (stores may be pending)
- complete memory operations: forward results and unblock waiting instructions
- complete other instructions: forward results and unblock waiting instructions
- update: write back results to physical register and update branch prediction tables
- graduate: remove completed instructions from active list, check for exceptions
- decode: remove instructions from fetch queue, decode, rename, determine dependencies and insert into active list
- fetch: access L1 instruction cache
- issue: send instructions to functional units or issue instructions to L1 data cache

2 Fundamental Data Structures

2.1 Dynamic Instructions

files: Processor/instance.h

The state of an instruction during execution is maintained in an *instance* data structure. This structure contains fields for the static instruction, such as logical source and destination register, instruction type and flags indicating the instruction class. The *instance* data structure adds to this the physical source and destination register numbers (after renaming), flags indicating the number and type of dependencies, the current and next PC, branch prediction information, exception status and statistical information. In addition, each instance has a unique tag value, which is incremented for every decoded instruction. This tag can be used to check the program order of instances with respect to each other. For fast memory management, instances are maintained in a pool within the processor model. The pool has a fixed size that corresponds to the size of the active list.

2.2 Activelist

files: Processor/active.cc, Processor/active.hh
Processor/active.h

The active list serves as instruction window and reorder buffer. Instructions enter the active list when they are decoded, and leave the active list during graduation. Each instance requires two entries in the activelist, since many instructions update two destination registers (general purpose and condition code register, or a pair of general purpose registers). For this reason, the activelist size is twice the number of active instructions.

2.3 Stall Queues

files: Processor/stallq.h, Processor/stallq.hh

Stall queues do not correspond to microarchitectural features of real microprocessors. They are used here to keep track of various dependencies and to maintain the order of instructions stalled for resources. Stall queues for physical registers are used to keep track of data dependencies. If during decode it is determined that an instance needs to wait for an operand to be produced by an earlier instance, it enters the stall queue for this particular physical register. Later, when the value is produced, the processor removes all instance from the stall queue for this register and checks if the instances are now ready to issue. Similarly, stall queues for functional units keep track of structural dependencies. When the instance is ready to issue but the functional unit is not available, the instance is added to that particular stall queue. Every time an instance frees up a functional unit, the next instance is removed from the stall queue and begins computing its result.

2.4 Heaps

files: Processor/heap.h

The processor simulator uses heaps to collect instructions during one cycle for processing in the next cycle. The running-heap contains all instructions that are currently executing in a functional unit. Instructions are added to the heap when they are issued to a functional unit, and the heap entry records the time when the instruction completes according to that functional units latency. Also at every cycle, the processor scans the running-heap for instructions that complete during this cycle and moves them to the done-heap.

Done-heaps are used to collect instances that have completed during a cycle. In the next cycle, the effects of these completed instructions are made visible to all other pipeline stages. The regular done-heap is used for all instructions except memory instructions. Instructions are added to the heap when they leave the running queue, that is when they leave the respective functional unit. In the following cycle, the results of all instructions in the done-heap are written to their destination registers, the branch outcome is compared with the predicted outcome and instructions that are waiting for the value produced by this instruction are removed from the stall queue.

The memory done-heap serves a similar purpose for memory instructions. Instructions enter the heap when the memory access completes in the cache, and are removed from the heap in the following cycle.

Note that these heaps do not correspond to a real hardware structure, they are only used for simulation purposes.

2.5 Processor State

files: Processor/procstate.h

The *procstate* structure (actually a C++ object) contains all data structures necessary to simulate a CPU. This includes various configuration parameters, the fetch and decode PC, arrays for the integer and floating point register file, copies of various supervisor state registers, lists of free rename registers, the rename map tables, branch prediction data structures, the active list, memory queues, running and done-heaps and the stall queues. In addition, it contains variables and data structures used to collect statistical information, such as the active list size, instruction count, branch mispredictions and stall times.

3 Parameters

files: Processor/config.cc

Most system parameters can be configured at runtime by the appropriate entries in the parameter file. The following table lists the parameters relevant to the processor model:

Parameter	Description	Default
activelist	number of instructions in activelist	64
maxaluops	number of integer instructions in the active list (if stall_on_full is set)	16
maxfpuops	number of FP instructions in the active list (if stall_on_full is set)	16
mamxmemops	number of memory instructions in the active list (if stall_on_full is set)	16
fetch_queue	size of instruction fetch queue	8
fetchrate	instructions fetched per cycle	4
decoderate	instructions decoded per cycle	4
graduationrate	instructions graduated per cycle	4
flushrate	instructions removed from activelist per cycle in case of an exception	4
bpbtype	branch prediction buffer type (twobit, twobitagree, static)	twobit
bpbsize	branch prediction buffer size	512
rassize	return address stack size	4
shadowmappers	number of shadow mappers for branch speculation	4
latshift	integer shift latency	1
latmul	integer multiply latency	3
latdiv	integer divide latency	9
latint	other integer instruction latency	1
latfmov	FP move latency	1
latfconv	FP conversion latency	4
latfdiv	FP divide latency	10
latfsqrt	FP square root latency	10
latflt	other FP instruction latency	3
repshift	integer shift repeat rate	1
repmul	integer multiply repeat rate	1
repdiv	integer divide repeat rate	1
repint	other integer instruction repeat rate	1
repfmov	FP move repeat rate	1

Table 4: Processor Runtime Configuration Parameters

Parameter	Description	Default
repfconv	FP conversion repeat rate	2
repfdiv	FP divide repeat rate	6
repfqrt	FP square root repeat rate	6
repflt	other FP instruction repeat rate	1
numalus	number of integer ALUs	2
numfpus	number of FP units	2
numaddrs	number of address generation units	1
nummems	number of load/store units	1
itlbtype	instruction TLB type: direct_mapped, set_associative, fully_associative	direct_mapped
itlbsize	total number of entries in instruction TLB	128
itlbassoc	instruction TLB associativity	1
dtlbtype	data TLB type: direct_mapped, set_associative, fully_associative	direct_mapped
dtlbsize	total number of entries in data TLB	128
dtlbassoc	data TLB associativity	1

Table 4: Processor Runtime Configuration Parameters

The *maxXXXops* parameters are only relevant if the processor model is compiled with the STALL_ON_FULL flag set. These parameters model a processor with separate issue windows for the different instruction classes, and cause the decode stage to stall if it attempts to issue an instruction to a full issue window.

4 Register File

Registers in ML-RSIM are dealt with in three levels. The first level is the architected view of the registers, or that which the instruction set directly sees. The architected registers are those such as, %g0-%g7, %i0-%i7, %g0-%g7, %o0-%o7 as well as other miscellaneous state and privileged registers. During instruction pre-decoding, the architected register numbers (as presented in the table below) are stored in the instruction object. When an instruction is decoded in the processor, this architected register is converted to a logical register via the arch_to_log() function.

Logical registers represent the full set of unrolled windowed architectural registers in the machine. These registers, with the exception of a few of the privileged registers are kept current with the architectural view of the fully unrolled set of SPARC registers. The logical view is only kept around as a simulator convenience, and is only needed to seed the physical register file during an exception. Physical registers represent the remapped physical registers in an out-of-order processor. This view is the view that is commonly accessed during normal instruction processing.

The global registers *(%g0-%g7*) are overlaid by a set of shadow registers when the CPU is in supervisor state (*pstate* = 1). This allows faster exception handling since the exception handler can work with a clean set of registers without saving them explicitly. Note however that only one shadow set exists, nested exceptions will use the same set and might interfere with each other.

In addition to the user registers, a number of supervisor state registers exist. Most of these registers are believed to correspond to the SPARC V9 architecture, however their usage might differ from the real implementation. Note that the first four supervisor state registers (*tpc* - *tt*) are implemented as a stack, with *tl* acting as stack pointer depending on the current trap level.

The following table summarizes the architected integer and state registers as well as the mapping to logical locations.

arch. #	Name	Description	logical #
0-7	%g0 - %g7	global registers	0-7 / 0,9-15
8-15	%i0 - %i7	input registers - windowed	
16-23	%l0 - %l7	local registers - windowed	
24 - 31	%o0 - %o7	output registers - windowed	
32-35	fcc0 - fcc3	floating point condition code	16 - 19
36 / 38	icc / xcc	integer condition codes	20
40	y	used by integer multiply/divide	24
42	ccr	condition code register	26
43	asi	current address space identifier	27
44	tick (read-only)	timer tick value	28
45	pc	program counter	29
46	fprs	floating point status register	30

arch. #	Name	Description	logical #
55	membar	memory barrier destination	39
56	tpc	trap PC	68+TL*4
57	tnpc	trap next PC	69+TL*4
58	tstate	trap state	70+TL*4
59	tt	trap type	71+TL*4
60	tick	timer tick value	40
61	tba	trap base address	41
62	pstate	processor state	42
63	tl	trap level	43
64	pil	processor interrupt level	44
65	cwp	current window pointer	45
66	cansave	number of savable windows	46
67	canrestore	number of savable windows	47
68	cleanwin	number of clean windows	48
69	otherwin	other windows	49
70	wstate	window state	50
71	fq	floating point queue	51
72	tlb_context	page-table base | faulting page #	52
73	tlb_index	TLB entry number for read/write	53
74	tlb_badaddr	faulting virtual address	54
75	tlb_tag	TLB entry tag portion	55
76	itlb_random	pseudo-random index for I-TLB	56
77	itlb_wired	upper limit of non-replacable entries for I-TLB	57
78	itlb_data	I-TLB entry data portion	58
79	dtlb_random	pseudo-random index for D-TLB	59
80	dtlb_wired	upper limit of non-replacable entries for D-TLB	60
81	dtlb_data	D-TLB entry data portion	61
82	tlb_cmd	TLB command register	62
83-86	-	unused	63-66
87	ver (read-only)	processor version	67

The *tick* (#44) registers can be read in user mode, it is incremented every clock cycle. The *FPRS* register implementation is compatible with the SPARC architecture. Bit 2 enables the floating point unit (in combination with the *fp-enable* bit in *pstate*), bit 1 indicates that the upper FP registers have been modified and bit 0 is set when the lower FP registers have been modified.

Registers starting at *tpc* can only be accessed in supervisor mode via *rdpr* and *wrpr* instructions. The processor version register is read-only, it returns the number of register windows in bits 7:0, the number of supported trap levels in bits 15:8 and a version number in bits 31:24.

Bit	Description	Power-up Value
31:6	unused	-
5	fp enabled	1 (enabled)
4	interrupt enable	0 (disabled)
3	alternate globals	0 (normal globals)
2	Data-TLB enable	0 (no address translation)
1	Instruction-TLB enable	0 (no address translation)
0	privileged mode	1 (privileged)

Table 5: PState Bit Definition

PState is the processor status register. Bit 0 indicates whether the processor is in user mode (0) or supervisor mode (1). Certain instructions (accesses to supervisor state registers, *DoneRetry*), are only legal in supervisor mode. Bits 1 and 2 enable (1) or disables (0) the address translation for instructions and data. Bit 3 controls which global register set will be used, and bit 4 enables interrupts. Bit 5 of the processor status register, in combination with the *FEF* bit (bit 2) in the *FPRS* register enables the floating point unit. The initial *PState* value is 0x0010 0001, thus enabling the FPU, disabling interrupts, the alternate globals and the TLBs while the processor is in supervisor mode.

PIL indicates the current interrupt level, it is initially 0 thus enabling all interrupts with a higher number. The TLB related registers are described in more detail in a later section.

4.1 Register Definition

files: *Processor/procstate.cc*
Processor/registers.h, Processor/procstate.hh, Processor/procstate.h

Most registers are implemented in conformance with the SPARC V9 architecture. To accommodate register windows, alternate globals and stacked trap registers, architected register names are first mapped to logical names that incorporate the current register window number and other state information affecting register mapping (routine *arch_to_log*). The following list shows how architectural names are mapped to logical names (before renaming):

- global registers: 0 - 7 or 0,9-15 depending on *pstate*
- non-privileged machine state registers (*fcc0* - *membar*): 16 - 39
- privileged machine-state registers (*tick* - *version*): 40 - 67

- stack (NUM_TRAPS deep) of trap state registers (*tpc, tnpc, tstate, tt*): 68 and up
- windowed register %ix, %lx and %ox: 68 + NUM_TRAPS*4 and up

The logical registers are then renamed to the physical registers via the intmapper and fpmapper constructs. New physical registers are allocated and freed via the appropriate freelist class, instantiated as proc.free_int_list and proc.free_fp_list.

4.2 Register Access

files: Processor/funcs.cc

All user-level state registers can be accessed using *wr* and *rd* instructions. These registers are renamed exactly like regular registers. Writes to *ASI* and *FPRS* have a global impact and are therefore serialized by setting the serialize-exception bit. In this way they get executed only when they are at the head of the reorder buffer and no other instruction is in the buffer.

The floating point status register (*FSR* or *XFSR*) is accessed with special load and store instructions. This register contains fields that may be modified by FP instructions, such as multiple condition code bits and other FP state. The condition code fields can be addressed explicitly by FP compare instructions, thus they are renamed like other general-purpose registers. In order to keep the renaming and dependency checking logic simple, instructions that read or write the FSR register raise a serialization exception if there are any floating point instructions ahead of the LDFSR/STFSR instruction at the time when it can be issued. The exception causes the instruction to re-issue, at which point it completes successfully.

Supervisor state registers are accessed using *rdpr* and *wrpr*. If the processor is in supervisor state, the read instruction executes like any other instruction (essentially the registers appear to be renamed), otherwise it raises a privileged exception. Write accesses are serialized similarly to writes to user-level state registers.

5 Little Endian Host Support

files: Processor/endian_swap.h, IO/byteswap.h

The SPARC architecture modeled in ML-RSIM is a big-endian architecture. The applications which are simulated are, obviously, also big-endian applications. Using a little endian architecture to run the simulator, such as the x86, presents a few special challenges. To support Linux/x86, endian conversions are required in circumstances where the native (x86) architecture needs to interpret or operate on simulated values.

To provide support for running the simulator on Linux/x86, values in memory are kept in the SPARC big-endian format. When values are loaded into registers, or are to be operated on or interpreted by the simulator, they are converted into the native host machine endianness. (Endian conversions are also required in a few places such as IO, where the simulator "looks at" simulated memory locations directly.) When values are stored from registers back into simulated memory, the endianness is converted back into the SPARC big-endian format.

For this, a set of functions are provided. Processor/endian_swap.h defines endian_swap, a set of C++ inline functions for swapping simulated values of various sizes to the native endianness. On big endian hosts, this effectively amounts to a NOP. On little endian hosts (currently only Linux/x86 is supported), this function does the appropriate endian swap for the size of the element being swapped. Since not all of the code in ml-rsim is written in C++, IO/byteswap.h defines a couple of macros. swap_short() and swap_word swap the endianness of a 16-bit and 32-bit word respectively.

Endian conversions may also be necessary when host memory is copied into simulated memory space, and via versa. This usually occurs in simulator traps, when a host side system call is performed to a task. In some of these traps, a struct is passed between the simulated kernel and the simulator via a simulated memory address. The simulator copies values into or out of this memory-based struct according to the definition of the trap. Where the values in the struct are produced or interpreted by the simulator, endian swapping must be performed.

6 Instruction Fetch Stage

files: Processor/exec.cc, Processor/memprocess.cc, Caches/cache_cpu.c
Processor/queue.h, Processor/fetch_queue.h

The instruction fetch stage issues fetch requests to the instruction cache. Instructions are fetched sequentially, unless the branch prediction logic in the decode stage or the graduation unit redirect the instruction stream. The instruction fetch unit communicates with the cache through two shared variables. The fetch PC is set by the instruction cache when it returns instructions to the fetch queue. It always points to the instruction that follows the last instruction in the queue. A separate flag indicates if the fetch PC is valid (that is, it has been set by the cache).

The decode stage (described in the following section) maintains the real PC, whereas the fetch PC is only speculative. The decode PC is updated by the branch prediction unit, the branch processing unit (the functional unit processing branches - an integer ALU) and the graduation unit (for exceptions). The decode unit compares the instructions in the fetch queue with the decode PC, and discards any instructions that don't match the expected PC. This feature models a processors ability to selectively discard instructions upon a branch misprediction. On the other hand, when detecting an exception, the fetch queue is flushed by removing up to *flushrate* instruction per cycle. In either case, when the decode stage detects that the fetch queue is empty it redirects the fetch PC.

Every cycle, the instruction fetch logic first checks if an external interrupt is pending. If this is the case and the interrupt-enable bit in the processor status word is set, it picks the highest priority interrupt from the pending-interrupt bit vector. If the current interrupt priority allows this interrupt, the routine inserts a *NOP* instruction with the appropriate exception level set into the fetch queue, and returns. This is repeated every cycle until the pending interrupt is cleared by the graduation stage.

If no interrupt is pending, the routine checks if the fetch PC is valid, and if the instruction cache is able to accept another request. If the instruction TLB is enabled, it then performs an I-TLB lookup. Only one TLB lookup is necessary per cycle, since instruction fetches never cross a cache line boundary, and hence never cross a page boundary. If the TLB lookup was not successful (TLB miss or protection fault) and the fetch queue is not full, a *NOP* instruction with the appropriate exception code set is inserted into the fetch queue. Otherwise, the fetch unit computes the number of instruction fetched in this cycle, determined by the maximum number of instruction fetched per cycle, and the alignment with respect to cache line boundaries. Note that instruction fetches are issued even if the fetch queue is full, since the decode stage may remove entries from the queue before the fetched instructions are returned from the cache. The fetch unit then calls the routine *ICache_recv_addr* which allocates a request structure and inserts it into the instruction cache request queue.

When data is returned from the I-Cache (in the following cycle, or later), the routine *PerformIFetch* is called. This routine inserts as many instructions as possible into the fetch queue and sets the fetch PC to the instruction following the last instruction in the queue. This indicates to the fetch stage from where to resume fetching, even if the cache was unable to insert all requested instructions into the queue.

The I-cache stores instructions as the static pre-decoded portion of an instance. A static instruction contains the op-code, source and destination registers and several other flags in an easy to process format. The routine *PerformIFetch* allocates a dynamic instance and copies the static instruction into the instance.

7 Instruction Decode Stage

files: Processor/exec.cc, Processor/branchpred.cc
Processor/exec.h, Processor/exec.hh, Processor/instance.h, Processor/branchpred.h

The instruction decode stage is mainly responsible for setting up the dependency control fields in the instance structure. It reads instructions from the fetch queue and compares the PC with the current PC. Unexpected instructions (whose PC does not match the decode PC) are discarded. If the fetch queue is empty (due to an I-cache stall, or after discarding all unexpected instructions), and the fetch PC is not equal to the next expected instruction (if it is, the fetch queue is empty because of an I-cache miss), the decode stage redirects the fetch stage by setting the fetch PC and the fetch synchronization flag.

An instruction that has been stalled in a previous cycle due to some resource conflict is kept in a separate pointer. Before reading new instructions from the fetch queue, the decode stage attempts to complete decoding of the stalled instruction. The actual decode process consists of the following two logical stages.

7.1 Instruction Decode

files: Processor/exec.cc

First, in the routine *decode_instruction* a dynamic instruction (an instance) is set up from the static description. This involves decoding the source and destination register type and finding the current rename mapping for the source registers. Then, the decode stage checks if the current instruction modifies the current register window (Save, Restore, Flush). If this is the case, it checks if a window overflow or underflow trap must be signaled. If the current instruction is a control transfer, the routine *decode_branch_instruction* attempts to determine the target address of the branch (through branch prediction, return address stack, or by calculating the target for unconditional branches). It also sets a flag indicating that the next instruction is the one that actually performs the control transfer.

7.2 Dependency Checking

files: Processor/exec.cc

The second decode phase (routine *check_dependencies*) takes care of all structural and data dependencies. It first attempts to allocate rename registers for the destination registers of the instance. If successful, it copies the old mapping into that instructions active list entry, and inserts the instance into the active list. If the active list is full, or no rename register is available, the routine sets a flag in the instance indicating at what point the decode process failed, and returns. The instance will be kept in the stall-pointer and the decode routine returns to it in the next cycle.

If the instruction is in the branch delay slot of a taken branch (or predicted taken branch), the current machine state is copied into a shadow mapper. The shadow mapper records the current register renaming state, it is used for fast branch misprediction recovery.

Next, data dependencies are checked. For every source register, the routine checks if the physical register is busy or not. If it is busy (value is not yet produced), the instance is added to that registers stall queue, otherwise the source operand is read into the instance structure. A flag indicates how many source operands have to be read, it is decremented for every available operand. If the current instruction access memory and one or more address calculation operands are not available, the address dependency flag is set, otherwise the instruction can be issued to the address calculation unit. In addition, memory instructions are entered into the memory queues (load or store queue).

At this point, if all true dependencies are met (data and structural), the instruction can be sent to the respective functional unit.

7.3 Context Synchronization

files: Processor/procstate.cc, Processor/exec.cc, Processor/exec.hh, Processor/memunit.cc Processor/procstate.h, Processor/instruction.h

Certain modifications to the processor state register, such as enabling or disabling address translation, require careful context synchronization. For instance, speculatively issued load instructions may bypass the *wrpr* instruction that enables the address translation. These loads can cause hardware (simulator) errors that should have been caught by the TLB. Context synchronization instructions prevent the decoding, dispatching and issuing of the following instructions until the synchronization instruction itself has graduated.

In the SPARC architecture, context synchronization is performed explicitly by the *Membar #Sync* instruction, and implicitly by the *DoneRetry* instruction which is used to return from an exception. When the processor encounters either instruction in the *check_dependencies* routine during decode, it sets the *SYNCtag* field to the tag of the synchronization instruction. This flag causes the current decode-loop to abort, and prevents any calls to *decode_cycle* in the following cycles, until the synchronization instruction graduates.

The graduation routine *remove_from_active_list* resets this flag when the synchronization instruction is graduated. In the following cycle, the processor will continue decoding and dispatching instructions.

In addition, the *SYNCtag* flag must be reset whenever the processor queues are flushed due to a mispredicted branch or exception. The routine *flush_active_list* removes all dynamic instructions from the processors active list (the reorder buffer). When it encounters a *DoneRetry* instruction, it resets the flag. Note that only one synchronization instruction can be outstanding at any given time. Memory barriers are handled in the memory queues, consequently, the *FlushMems* routine detects the synchronization memory barrier instruction and resets the flag.

8 Instruction Issue and Completion

8.1 Issue

files: Processor/exec.cc
Processor/exec.h, Processor/exec.hh, Processor/stallq.h, Processor/stallq.hh

Instructions are issues as soon as all operands are available and the functional unit is not busy. This is controlled by counters in the instance structure that indicate how many data dependencies have not been satisfied, and by the stall queues associated with each physical registers and the functional units.

Whenever an instruction produces a result and the result is written to a physical register, the busy bit of this registers is cleared and all instances are removed from the respective stall queue (routines *CompleteQueues*, *update_cycle* and *ClearAll*). For every instance in a particular stall queue, the routine *ClearAll* clears the respective dependency bit and issues the instruction if both data dependencies and structural dependencies are satisfied.

Similarly, when a functional unit is available, the next instance is removed from the stall queue (if the data dependencies are satisfied), the result is computed and the instance is scheduled to complete after N cycles (N is the FU latency).

Instances that are currently executing are kept in a running heap. This heap does not correspond to a real microarchitectural feature, it is used only to simplify the completion stage. Each entry consists of an instance pointer and the time when the instruction is scheduled to complete.

8.2 Completion

files: Processor/exec.cc
Processor/exec.h, Processor/exec.hh

In every cycle, the processor simulator scans the running-heap for instructions that are completing this cycle. Instructions that complete are moved to the done-heap. This heap is only used to communicate between pipeline stages. In the same cycle, the routine *update_cycle* removes instructions from the done-heap and writes back the results to the physical registers.

In addition, for branch instructions the actual branch outcome is compared with the predicted branch target. Upon a misprediction, the register map tables are restored from the shadow mapper, and the active list, memory queue, fetch queue and stall queues are flushed. On a correct prediction, the shadow mapper is deallocated so that it is available for other branch instructions.

Instructions that completed without exception are marked as done in the active list.

9 Instruction Graduation Stage

files: Processor/active.cc

The graduation stage is responsible for removing completed instructions from the active list while checking for exceptions. In every cycle, the routine *remove_from_active_list* checks the completion and exception status of the *graduationrate* oldest instructions in the active list.

If an instruction encountered an exception, the routine returns with a pointer to the instruction. The exception handling code then flushes the active list, fetch queue, all stall queues and resumes execution at the appropriate exception vector address.

If the graduating instruction is a context synchronization instruction (*DoneRetry* or *Sync Barrier*), the synchronization flag is reset which causes the decode stage to resume. If the instruction has completed without exception, it is removed from the active list and various statistic fields are updated.

Another responsibility of the graduation stage is to mark non-speculative memory operations as ready to issue in the following cycle. This is necessary for store instructions and uncached loads, since these instructions can only be issued non-speculatively. The routine *mark_memops_ready* is called after *remove_from_active_list*, it scans the next N instructions in the activelist (instructions that may graduate in the following cycle). The scan aborts as soon as either *graduationrate* instructions have been examined, or an instruction is flagged with an exception (since following instructions will not graduate).

If a store that is ready to issue (address has been calculated) is encountered, the instruction is marked as ready to issue to the memory hierarchy. In addition, a copy of the instance is made which replaces the original instruction in the store queue, and the original instance as marked as completed in the active list. This allows the store to graduate before it actually has been issued to the caches. The instance copy is needed in the memory unit because the original instance is deallocated when it graduates. Uncached loads are only marked as ready to issue to the cache, since they can not graduate before the load value is returned.

10 Memory Instruction

Due to their special characteristics, memory operations are handled slightly differently from other instructions. Upon decode, the instance is inserted both into the active list and the load or store queue. Address generation is handled by the normal dependency checking logic with stall queues. Once the address is calculated and the translated to a physical address, loads may issue but stores have to wait until they are about to graduate.

10.1 Memory Instruction Issue

files: *Processor/memunit.cc, Processor/active.cc*
Processor/memunit.h, Processor.memunit.hh

Every cycle, the routine *IssueMem* attempts to issue ready load and store instructions. First, it checks if a memory barrier can be removed from the queue of pending barriers. The processor maintains a list of barriers with their instance tag. If the oldest load or store is younger than the oldest memory barrier (depending on the barrier type), the memory barrier can be removed from the queue. The routine then tries to issue store and loads. Stores are given priority because they are issued only non-speculatively, and are generally older then loads.

The routine *IssueStores* scans the store queue for instructions that are ready to issue (address is ready, marked as non-speculative), but have not been issued yet. For every such instruction, it checks if there exists an older store with an overlapping address. If this is the case, the store instruction can not be issued. Otherwise, the routine checks if the cache port is available, and sends the instruction to the cache.

The routine *IssueLoads* searches through the load queue for unissued loads. For every such instruction it searches the store queue for older un-issued stores. If an older store instruction has its address calculated, and the addresses overlap, the data can either be forwarded from the store, or the load has to stall (if forward would cross a memory barrier). If no conflict is found, and the cache port is available, the load instruction can be issued to the cache.

10.2 Memory Instruction Completion

files: *Processor/memunit.cc, Processor/memprocess.cc*

Memory instructions complete when the data is read or written at the L1 cache. The cache calls the routines *PerformData* and *MemDoneHeapInsert* to perform the memory access and to signal to the processor that the instruction has completed.

PerformData calls the actual memory access function associated with the instruction. These routines look up the host address corresponding to the simulated physical address in the simulator page table, and perform the respective load or store.

The routine *MemDoneHeapInsert* inserts the completed instance into the memory done-heap. Similar to the normal done-heap, this is a data structure that collects all completed instructions for processing in the same cycle.

The routine *CompleteMemQueue* processes all instance in the memory done-heap that have completed in this cycle. For every such instruction, it calls *CompleteMemOp*, which in turn calls *PerformMemOp*. In case of a store it also deallocates the instance, since it was allocated as a copy of the original instance when it was marked as ready to issue.

Load instructions need to be handled differently, since they are issued speculatively. At the time of completion, the routine *CompleteMemOp* checks if there are any ambiguous stores (address has not been calculated) that are older than the load. If this is the case, the *limbo* count of the load is incremented. Otherwise, if an older store is found which overlaps with the load and the data has not been forwarded when the load was issued (load tag = – store tag), the load needs to issue again. In any other case (no conflicting store, or ambiguous stores found), the load is speculatively completed by calling *PerformMemOp*.

The routine *PerformMemOp* simply removes complete stores from the store queue. In case of a load, the routine checks if the load is preceded by ambiguous stores. If this not the case, and the load does not need to be reissued (it is not marked as complete), the instruction is removed from the load queue. Otherwise, the load is still speculative and is not removed from the queue, while the load value is written to the destination register where it is available for dependent instructions.

10.3 Memory Disambiguation

files: Processor/memunit.cc

Memory disambiguation is performed in two places. When a load instruction is ready to issue, it checks if it is preceded by ambiguous stores or stores that overlap with the load. If this is not the case, the load is not speculative with respect to data dependencies. If an older store is found that overlaps with the load, the data value can be forwarded and the load is marked as forwarded by setting its tag to the negative tag of the store instruction. When the load completes, it again checks if there are any older stores that overlap. If this is the case, and the value had been forwarded correctly, the load can complete. If a younger store is found that overlaps with the load, the data value can be forwarded at this point. If, however, any of the preceding stores has not had its address calculated, the load is considered in limbo, and a flag is set to indicate this.

Whenever a store instruction calculates its address, it checks the load queue for any loads that are in limbo state. If the load indeed conflicts with the store, the load is marked with the *limbo* exception, which causes it to re-executed when it is ready to graduate. Otherwise, the limbo count of the load is decremented, and the load is again performed by calling *PerformMemOp*. This routine removes the load from the load queue if the limbo count is zero, otherwise it just updates the destination register with the load value.

10.4 Address Generation

files: Processor/memunit.hh, Caches/cache_cpu.c
Processor/instance.h

The routine *CalculateAddress* is called by the out-of-order execution engine when the address unit is ready to accept another instruction, and the address dependencies have been satisfied for this instruction. It first calculates the effective address and then performs the data TLB lookup, if it is enabled in the processor status word. If the TLB lookup is successful, the memory attribute field of the instance is set and the instruction continues execution normally. Otherwise, the instance is marked with a *DATA_FAULT* or *DTLB_MISS* exception.

10.5 Uncached Load/Store Instructions

files: Processor/memunit.cc, Processor/procstate.cc, Processor/active.cc
Processor/procstate.h, Processor/procstate.hh

Uncached load and store instructions have to issue strictly in-order and non-speculatively. In this respect, they are similar to regular store instructions. A store instruction in the memory queues can issue only when it is in the set of instructions that will be retired in the next cycle and if no preceding instruction causes an exception. The function *mark_memops_ready* is called every cycle and marks both uncached loads and all stores as ready if they will be retired in the following cycle.

Uncached stores are handled no different than cached stores. However, uncached loads can only issue if they have been marked ready, thus preventing speculation of uncached loads. Furthermore, data forwarding is disallowed between uncached stores and subsequent loads.

The function *IssueLoads* is called every cycle to issue load instructions to the memory hierarchy. The rules under which a load can issue prevent that an uncached load is not issued past a memory barrier.

10.6 Memory Barrier & Uncached Buffer

files: Processor/memunit.cc, Caches/cache_cpu.c
Processor/memunit.h, Processor/memunit.hh, Caches/cache.h

In order to model the fence effect of a memory barrier in a stream of uncached loads or stores, it is necessary to issue the memory barrier instruction to the uncached buffer. *IssueBarrier* is called when the memory unit determines that a memory barrier can be broken down, i.e. when all previous memory operations have been issued. This function checks if the queue to the uncached buffer is full, if not it calls *DCache_recv_barrier* which forms a request of type BARRIER and inserts it in the queue. In addition, a memory barrier instruction of type *MemIssue* does not graduate until the uncached buffer and the queue to the system bus are empty.

11 Exception & Interrupt Handling

Exception handling is modelled after the SPARC V-9 architecture specification. The simulator implements a trap table in unmapped memory and supports all supervisor state registers needed to handle traps in software. The following table summarizes the supported exception classes in order of decreasing priority:

Code	Exception	Description
0	OK	no exception, instruction can graduate
1	Serialize Instruction	certain instructions that modify or access global state e.g. DoneRetry, wrpr, rdpr these exceptions are handled in hardware
2	ITLB Miss	instruction TLB miss
6	DTLB Miss	data TLB miss
10	CSB Flush	CSB flush executed in user code
11	Privileged Instruction	privileged instruction encoutered in user code
12	Instruction Fault	instruction address translation fault, translation invalid or access protection fault
13	Data Fault	data address translation fault, translation invalid or access protection fault
14	Bus Error	misaligned memory access
15	Illegal Instruction	undefined opcode
16	Clean Window	Save instruction needs to clean a window
20	Window Overflow	Save/Flush instruction needs to save next window
24	Window Underflow	Restore instruction needs to restore previous window
28	Soft: Limbo	load misspeculation, load conflicts with store, handled in hardware
29	Soft: Coherency	load misspeculation, cache line was replaced due to coherency before load graduates, handled in hardware
30	Soft: Replacement	load misspeculation, cache line was replaced due to cache conflict, handled in hardware
31	FP disabled	floating point operation while FPU is disabled
32	FP Error	floating point unit exception
33	Divide by Zero	division by zero (integer only)
34-81	System Trap 00 - 2F	system call traps
82-97	Interrupt 0F - 00	external interrupts

Table 6: Exception Codes

System trap 0x02 is designated as *simulator trap*, it is used to communicate directly with the simulation host operating system or the simulator. Solaris normally uses trap 0x08, and occasionally uses trap 0x27. An external interrupt of type 0 has lowest priority.

11.1 Trap Table

files: Processor/except.cc, Processor/active.cc
Processor/instance.h

The trap table is a structure in unmapped memory that contains the entry points for all supported traps and exceptions. The trap-base address register *TBA* contains the base address of the trap table. The entry point of an exception handler is formed by adding the exception number shifted left by 5 bits to the trap table base address. Each exception handler entry point in the table can store 8 instructions. For a few critical traps, the exception codes are set up such that the following entry points are unused, leaving space for up to 32 instructions. This allows inlining of performance critical exceptions such as the TLB miss handlers and window traps.

11.2 Exception Detection

files: Processor/funcs.cc, Processor/memunit.hh, Processor/signalhandler.cc

Exceptions are flagged in the *instance* structure as well as in the corresponding active list element. Most exceptions are detected during instruction execution. For instance TLB traps are detected when the effective address is calculated and the virtual address is looked up in the TLB. Similarly, privileged, illegal or serialize instruction traps are detected when the instruction is issued to the functional unit. Instruction fetch exceptions (I-TLB miss etc.) and external interrupts are detected in the instruction fetch stage, in which case the fetch unit inserts a NOP instruction into the pipeline with the exception flag set.

Floating point exceptions are flagged by a SIGFPE signal handler. Before a floating point instruction is executed, the emulation routine sets the hosts FP trap mode to the trap mask of the simulated processor. If an exception occurs while the floating point instruction executes, the signal handler marks the exception in a global variable. The emulation routine checks this variable and flags an exception if it is set. The detailed exception value is stored in the *rscc* element of the instance.

11.3 Exception Handling

files: Processor/except.cc

ML-RSIM implements two types of exceptions. Serialized instructions do not need to trap to an exception handler, they can be simulated in a single cycle after the pipelines and queues have been flushed. The function *ProcessSerializedInstruction* handles reads and writes to user-level control and status registers (such as *TICK*), privileged reads and writes to supervisor state registers, serialized multiply instructions, loads and stores to the floating point status register, window save and restore instructions and *DoneRetry* instructions. Similarly, *soft* exceptions do not trap to software but simply cause the trapped instruction to issue again to resolve the conflict.

The remaining exceptions are handled by a software trap handler. These exceptions include TLB misses, segmentation faults, window traps, system call traps and external interrupts. The routine *SaveState* increments the trap level register *TL* and then copies the current *PC*, *NPC* and *PState* register into *TPC[TL]*, *TNPC[TL]* and *TState[TL]*, respectively. It also copies the condition code register into bits 31:24 of *TState[TL]*. It then switches into privileged state, enables the alternate globals, disables all interrupts and writes the interrupt type into the *TT* register. In addition, it disables the instruction TLB for all exceptions and the data TLB for TLB miss traps. After that, the routine computes the trap handler address by adding the exception code shifted left by 5 bits to the trap table base address.

The *RetryDone* instruction is used to return from an exception handler, it is only allowed in supervisor mode. *Retry* should be used if the faulting instruction should be restarted, for instance after the TLB miss handler installed the correct mapping into the TLB. *Done* is used if the instruction should not be re-executed, for instance when the exception handler emulated the instruction. Both instructions are serialized and restore the processor state by writing back the trap shadow register contents into the original registers and decrementing the trap level *TL*.

11.4 System Call Traps

files: Processor/funcs.cc, Processor/except.cc
Processor/instance.h

The immediate value of the *trap* instruction (or the source register value) determines which system trap is triggered. Trap 0x02 is used for simulator traps, which are used to access the underlying operating system, for instance to open files or allocate physical memory. Trap 0x08 is the standard Solaris system call interface. In addition, Solaris occasionally uses traps 0x24, 0x25 and 0x27, probably for backwards compatibility.

11.5 External Interrupts

files: Processor/active.cc, Processor/procstate.cc, Processor/memprocess.cc
Processor/procstate.h, Processor/memprocess.h

Each processor contains a variable *interrupt_pending* that is a bit mask of pending external interrupts. The routine *fetch_cycle* checks if any interrupts are pending before issuing a new fetch request to the cache. If the interrupt bit vector is not 0 and the interrupt-enable bit in the processor state register is set, it determines the pending interrupt with the highest priority and inserts a NOP instruction with the correct interrupt level set into the fetch queue. This is repeated once per cycle until the exception is taken. The pending interrupt bit is cleared in the exception handling routine when the interrupt is taken.

The simulator catches the SIGINT signal (ctrl-C) and forwards it as external interrupt 1 to all processors, which allows the simulated OS to gracefully shut down if the user wants to interrupt the simulation.

11.6 Statistics

files: Processor/procstate.cc, Processor/except.c, Processor/active.cc
Processor/procstate.h

The exception handling routine and the graduation stage count the number of instructions and cycles for each exception level. This provides a detailed analysis of how much time the processor spends at each exception level.

The processor maintains an array of instruction counts and cycle counts for each exception level. The graduation stage increments the counter corresponding to the current level for each instruction that graduates. In addition, it counts the number of memory references that graduate while the MMU is enabled. This number is used to calculate the TLB hit rate.

Whenever an exception is taken, the exception handling routine stores the cycle number in an auxiliary array. Upon return from the exception, it computes for how many cycles the processor was handling this exception, and adds the result to the array of exception cycle counts. Note that this count includes the number of cycles that the processor spends handling nested exceptions of a higher priority. For instance, if the window overflow handler encounters a TLB miss, the window overflow handler cycle count includes the number of cycles spend handling the TLB miss.

For any exception that occurred during execution, the processor statistics routine prints out the number of exceptions, the number of instructions and number of cycles spent handling it as well as the resulting IPC.

12 System Calls and Simulator Traps

files: Processor/traps.cc

This chapter describes the system call interface and the way system calls are handled by the simulator. To make the distinction between true system calls and simulator traps more clear, a new exception type *SIMTRAP* has been introduced. These exceptions are not simulated but complete in one cycle. They are used to communicate simulator configuration parameters to the runtime system (like the TLB type), or to perform operations that can not be simulated (like file accesses).

12.1 System Call Interface

The Sun Solaris system call interface specifies that the system call number is stored in register %g1. Registers %o0 and above are used for arguments, the result is returned in %o0. The immediate value of the *unimp* or *tcc* instruction indicates the trap handler. Solaris normally uses trap 0x08 and occasionally system traps 0x24, 0x25 and 0x27. Trap 0x02 is used for simulator traps, which are essentially system calls that are passed to the simulation host operating system.

12.2 Simulator Trap Handling

files: Processor/except.cc, Processor/traps.cc

When the graduation stage encounters a *SIMTRAP* exception, it drains the pipelines like in the case of any other exception. The exception handling routine then first calls *SimTrapHandler* which executes all exceptions which are not actually simulated, like host file accesses, physical memory allocation or statistics control. If this routine does not find an applicable trap handler, it returns 0 and sets the exception code of the instruction appropriately, which causes the exception handling routine to trap to the registered simulated exception handler.

12.3 Simulator OS Bootstrap Information

12.3.1 Get Segment Info

This supervisor-mode simulator trap reports the beginning and end of the text, data and stack segments. It is used by the runtime environment during initial page table setup, since the executable is loaded by the simulator and not the runtime system. The argument specifies whether the lower/ upper bound of the text or data segment is requested. The call returns a page aligned physical address.

12.3.2 Get STDIO

This trap is used to communicate the stdio file names to the operating system, since the names may be specified at the simulator command line by the user. The call takes a file number (0, 1 or 2), a buffer pointer and the buffer size as arguments and copies the requested filename into the buffer space. It returns 0 on success or *-EINVAL* if the requested file descriptor is invalid. In addition to these filenames it can also provide the initial working directory and the root directory for simulated disk mount points, which is determined relative to the location of the simulator executable.

12.3.3 Get UID and GID

During the simulator boot process, the OS changes the user and group ID of the first user process to the identity of the user running the simulation, so that the simulated application has access to the user files on the host system. The two simulator traps *GetUID/GetGID* return the user and group ID of the user running the simulator.

12.3.4 Get Groups

Similarly to the *GetUID/GetGID* traps described above, this trap returns the list of groups to which the simulator user belongs, so that the simulator OS can change the credentials of the simulated user process accordingly. The trap takes the maximum number of group elements and a pointer to a group vector as arguments. It returns the group vector in the buffer provided.

12.4 File I/O Simulator Traps

files: Processor/traps.cc

These simulator traps allow simulated applications to access user files on the host system. These calls are mainly used by the *hostfs* filesystem which is part of the simulator OS. The following table lists all file system I/O related traps with a short description:

Trap	Arguments	Description
DCreat	dir-fd, name, mode	change to dir-fd and create directory
DOpen	dir-fd, name, mode, flags	change to dir-fd and open file
PRead	fd, buf, size, offset	read size bytes into buf, start at offset
PWrite	fd, buf, size, offset	write size bytes from buf, start at offset
LSeek	fd, pos, whcnce	change current filepointer
Dup	fd	duplicate file descriptor
Readlink	link, buf, size	read link into buf
Close	fd	close file descriptor
DUnlink	dir-fd, name	change to dir-fd and delete file
DAccess	dir-fd, name, mode	change to dir-fd and check access privileges to file
Ioctl	fd, cmd, arg	do special operations on fd
FCntl	fd, cmd, arg	do special operations on file
FStat	fd, buf	read file status into buf
DFStat	dir-fd, name, buf	change to dir-fd and read status of file name into buf
LStat	fd, buf	read status of fd into buf but follow links

Trap	Arguments	Description
Poll	pollfd, nfds, timeout	poll on a set of file descriptors
Chmod	name, mode	change permissions of file name
FChmod	fd, mode	change permissions of file fd
Chown	name, uid, gid	change uid and gid of file name
FChown	fd, uid, gid	change uid and gid of file fd
LChown	fd, uid, gid	change uid and gid of file fd follow links
DMkDir	dir-fd, name, mode	change to dir-fd and create directory name using mode
DRmDir	dir-fd, name	change to dir-fd and delete directory name
GetDents	fd, buf, size	read directory entries from fd into buf
Link	oname, name	create link from name to oname
SymLink	oname, name	create symbolic link from oname to name
FStatVfs	fd, buf	read file system status into buf

A capital D in the trap name indicates that the first argument is a directory file descriptor. This is used as base directory for traps that take file names or directory names as arguments. Since the simulator may run multiple programs with independent current working directories, it uses the directory file descriptor argument to first change into that directory before performing the desired function.

Many of these simulator calls operate on physical addresses. The simulated caller must therefore translate buffer addresses into physical addresses before invoking the simulator trap. In addition, the caller must ensure that buffers do not cross page boundaries since contiguous addresses in virtual space do not necessarily correspond to contiguous physical addresses. A memory barrier followed by a non-faulting memory instruction should be used to make sure that all outstanding memory references to the buffer space have completed.

The *stat* system calls take a pointer to a structure as argument. The simulator trap copies the elements of the native stat-structure to the application-provided structure. However, if the simulator is running on a non-Solaris architecture, the layout and size of the individual elements might be different. For this reason, the simulator traps copy the structure element by element, and issue a warning if the element size in the native structure differs from that of the application. Similarly, the *fcntl* system traps convert between the native flock structure of the simulation host and the Solaris/Lamix format.

Traps handlers operating on file descriptors use the simulator file descriptor management facility to reduce the number of concurrently open files. When a file is initially opened, an entry is allocated in a table that contains the filename, access mode and current file offset. The entry index is returned as a file descriptor and the file is closed. Subsequent simulator traps that take a file descriptor as an argument use the file descriptor to index into the table, open the file with the original access mode, seek to the most recent file offset and perform the desired operation. Once complete, the file is closed again. When a file is explicitly closed by the simulated software, its entry is removed from the table. In this way, the simulator can support a virtually unlimited number of 'open' simulated files without exhausting its file descriptor table.

12.5 Socket Simulator Traps

files: Processor/traps.cc

Socket-related traps allow simulated software to establish socket connections with processes running outside the simulator. Currently, these traps support only Internet domain sockets. When establishing a socket, the *socket* trap handler converts the Solaris/Lamix domain and protocol identifier into the native representation. The file descriptor returned by the system call is forwarded to the simulated software as a return value. Unlike regular file descriptors, the simulator leaves socket descriptors open until they are closed explicitly by the simulated kernel.

Simulator traps dealing with socket addresses (*bind*, *connect*, *accept*) convert between the Solaris/Lamix representation and the native format, while also adjusting the address family specifier. The *sendmsg* and *recvmsg* traps are the only mechanism to read or write data on a socket. Simulated software uses these traps with different parameters to implement all flavors of data transfer system calls. Currently, these traps do not support the *accrights* component of a message structure. Sockets are closed by a special simulator trap, because unlike regular files, socket descriptors correspond to actual open files in the simulator executable and must be closed.

The socket option traps convert the option name between Solaris/Lamix and native format. The option value can only be converted correctly if it is either a *linger* structure or a 32 bit integer value.

To support applications relying on dynamic host lookups via the *gethostbyname* or *gethostbyaddr* routines, two corresponding simulator traps are provided. However, instead of the native *hostent* structure, a flat, non-dynamic equivalent structure is used between the kernel and the simulator trap handler to simplify virtual-to-physical address translation as required by the trap handler.

12.6 Statistics Simulator Traps

The simulator provides several traps that control statistics collection and reporting. These traps are not privileged, so that user level software may use them if necessary.

The *clear_stat* trap resets all statistics counters and other related structures for all modules in the system, including all processors (not only the calling processor). Similarly, the *report_stat* call triggers a printout of the current statistics values for all modules.

User statistics objects are allocated by the *userstat_alloc* trap. This trap takes the statistics type and name as arguments and returns the index of the object which is used to subsequent userstat*_sample* traps. Note that the name must be provided as physical address and must not cross a page boundary User statistics sample are triggered by the *userstat_sample* trap. The semantics of this trap depend on the particular statistics type.

12.7 Miscellaneous Simulator Traps

12.7.1 Allocate Page

This simulator trap supports a simple physical memory management system. Each node maintains its own simulator page table, which maps simulated addresses to host addresses. Note that this page table is different from the Lamix operating system maintained page table, it is transparent to the simulated software. The trap allocates a page of host memory and inserts an entry into the simulator page table, which maps a specified simulated address range to this host address. This allows the simulated operating system to use arbitrary pages and map them into application address space, while making sure that the simulator is able to translate these addresses to actual host addresses. This trap is only available in supervisor mode.

12.7.2 Write Log

This simulator trap gives the simulator operating system the ability to write information to the simulator log file '<subject>.simout'. The trap takes a buffer pointer and the buffer length as arguments, writes the buffer contents to the log file and flushes the file so that any changes are immediately visible. Note that the buffer address is a physical address, the OS needs to perform an address translation before calling this trap.

13 Instruction & Data TLB

By default, the processor model provides independent data and instruction TLB. However, by setting either TLB size to 0 a unified TLB is modeled. Three different TLB types can be simulated. The set-associative TLB supports true LRU replacement in hardware while the fully-associative TLB relies on software support for random replacement (or any other strategy).

When an instruction that encountered a TLB miss reaches the graduate stage, the processor flushes the pipeline as it would for any other instruction. Subsequently, it copies the virtual address into the *badaddr* register, copies the page number into the lower bits of the *context* register and initializes the *index* register. Finally, the TLB miss handler is called. If the TLB is configured as fully-associative, the handler needs to perform the following steps:

1. read *badaddr* or *context* to perform the page table walk
2. read *random* and write its content into *index*
3. write *tag*
4. write *data* - this will trigger the TLB write operation at *entry[index]*

The *context*, *index*, *badaddr* and *tag* registers are shared between both TLB, since only one TLB exception is handled at any given time. On the other hand, the *random*, *wired* and *data* register are separate for each TLB, because they contain TLB specific data and writes update a specific TLB. The *random* register provides a pseudo-random value between the value of *wired* and the total number of TLB entries. Its value can be used for pseudo-random replacement of TLB entries. Software can however choose to write any other value into *index*, for instance to change the wired entries that are located between entry 0 and *wired*-1. If the TLB is configured as set-associative or direct-mapped, the TLB handler performs the same steps as above with the exception of step 2. It is not necessary to write the *index* register since hardware writes the correct index when the processor takes the TLB miss exception.

13.1 TLB Configuration

files: Processor/config.cc, Processor/tlb.cc
Processor/tlb.h

The following table summarizes the TLB configuration parameters.

Parameter	Description	Default
itlbtype	direct_mapped, set_associative, fully_associative, perfect	direct_mapped
itlbsize	total number of entries	128
itlbassoc	associativity	1
dtlbtype	direct_mapped, set_associative, fully_associative, perfect	direct_mapped
dtlbsize	total number of entries	128
dtlbassoc	associativity	1

TLB size refers to the total number of entries, regardless of set associativity. Associativity is only relevant for the set-associative configuration, it is ignored otherwise. Setting either TLB size to 0 (but not both) specifies a unified TLB. A perfect TLB is modeled by handling all TLB misses instantaneously, thus the specified TLB size may affect simulator performance.

13.2 TLB Entry Format

files: Processor/tlb.cc
Processor/tlb.h

Each TLB entry contains a 32 bit tag, a 32 bit data field and an age counter for LRU replacement in case of a set-associative TLB. The following table shows the individual bits. A 1 means that the attribute is set

Bits	Description
tag <31:11>	virtual page number
tag <10:1>	unused
tag <0>	entry valid
data <31:11>	physical page number
data <10>	wired (not used by hardware)
data <9-6>	unused
data <5>	privileged
data <4>	non-coherent
data <3>	uncached accelerated
data <2>	uncached
data <1>	read only
data <0>	mapping valid

Table 7: TLB Entry Definition

The entry-valid bit indicates only if the tag portion of the entry is valid, it is used during a TLB lookup. An invalid entry will not be considered for tag comparison.

The attribute bits in the data part are passed to the instruction instance and the request structure as attributes. They are used to distinguish normal form I/O space accesses, and data from instruction accesses. The mapping-valid field indicates whether the particular mapping is a valid one. If a TLB lookup operation encounters a valid entry (entry-valid is set) but with an invalid mapping, a TLB fault exception is triggered. For example, if an application accesses an illegal address, it will first encounter a TLB miss. The TLB miss handler will be unable to find a valid mapping in the page table and fill the TLB with an invalid mapping. The retried memory operation will then trigger a segmentation fault.

13.3 TLB WriteEntry

files: Processor/except.cc, Processor/tlb.cc
Processor/tlb.h

A TLB entry is written by a *wrpr* instruction with *itlb_data* or *dtlb_data* as target register. The *index* register points to the TLB entry that should be written and the *tag* and *data* registers contain the tag and data portions of the TLB entry.

The operation of the *WriteEntry* method depends on the TLB type. For direct-mapped or fully-associative TLB, the method simply writes tag and data values into the entry that the index parameter points to. In case of a set-associative TLB, the method needs to find the least recently used entry before writing tag and data. It first looks for an invalid entry within the set, if none is found it looks for the entry whose age is 0.

13.4 TLB LookUp

files: Processor/memunit.cc, Processor/tlb.cc
Processor/memunit.hh, Processor/tlb.h

An instruction TLB lookup is performed at most once per cycle in the fetch stage. Since instruction fetches do not cross cache line boundaries, it is not necessary to perform a lookup for every fetched instruction. The data TLB lookup is performed after the address calculation stage. The function *CalculateAddress* first calculates the effective address (in *GetAddr*) and then calls the TLB *LookUp* method.

The *LookUp* method takes as parameters a pointer to a virtual address (which will be replaced by the physical address), a pointer to a page attribute specifier, flags indicating if the access is a write and if the processor is in privileged mode, and the tag of the instance that performs the TLB lookup. The routine returns 0 if the address translation was successful, 1 upon a TLB miss and 2 upon an access protection fault (write to read-only space, access to privileged space in unprivileged mode, or access to a non-mapped page). The calling routine is responsible for performing the TLB lookup only when the respective TLB is enabled in the processor status word, and for marking the instruction with the appropriate exception code.

In case of a fully-associative TLB, the method searches the entire TLB sequentially for a matching tag. Upon a hit, it replaces the page number portion of the instance address with the new page number. To speed up fully-associative lookups, the TLB maintains a pointer to the most recently accessed entry, and starts searching from this entry for subsequent lookups. For a set-associative TLB, the lookup method first needs to calculate the correct set index. It then searches through the set for a matching and valid entry. If such an entry is found, it replaces the page number portion of the instance address with the new page number. The matching entries age is set to the maximum value (*associativity*- 1). The age of all entries within the set with age higher than the original age of the matching entry is decremented.

In case of a direct mapped TLB, the method simply calculates the page number and checks if that entry is valid and matches the virtual page number.

Unless a perfect TLB is modeled, if no matching and valid entry is found, the method returns with a status indicating a TLB miss. If, however, a valid tag was found but the mapping-valid bit is cleared, or a write to a read-only page was attempted, or a non-privileged access to a privileged page was attempted, a *tlb_fault* status is returned.

A perfect TLB is modeled as a fully-associative TLB with instantaneous fills upon a miss. The TLB lookup algorithm is identical to that of the fully-associative TLB, but when no matching entry is found, the lookup routine itself attempts to perform a TLB fill. It reads the current process context (the page table root pointer) and adds the upper 10 bits of the access address to it to read the pointer to the level-1 page table. If that entry is invalid, a TLB fault is returned, otherwise, that pointer is combined with the middle 10 bits of the access address to locate the correct page table entry. Then, the entry is written into the TLB and a TLB hit or TLB fault (write to read-only page, or user-access to privileged page) is returned. Each access to physical memory must also check the list of pending stores for a conflict, in which case the most recent store value is read instead of the value in physical memory. This is important when a newly allocated and mapped page is accessed shortly after the page table entry has been updated, since the page table modification may still be buffered in the processor store buffer, or may be proceeding through the caches.

13.5 TLB Probe and Flush

files: Processor/except.cc, Processor/tlb.cc
Processor/tlb.h

Writing to the *tlb_cmd* register triggers special operations like probe and flush. Writing a 0x01 (0x10) results in an instruction TLB (data TLB) probe operation, which looks for a valid entry that matches the tag in the *tlb_tag* register and returns the entry number in *tlb_index* upon success. If no matching entry is found, bit 31 of the index register will be set to 1.

Writing the value 0x02 (0x20) to the command register clears all instruction TLB (data TLB) entries.

Part III: Memory Hierarchy

1 Overview

A node consists of one or more processors, each with its associated L1 instruction/data cache and L2-cache, a system bus, main memory controller and a collection of I/O devices. Bus masters are numbered consecutively within a node, starting at CPU 0. I/O devices start at *numcpus*, with coherent I/O devices (devices that snoop bus transactions) being first, followed by non-coherent I/O devices. The memory controller is the highest-numbered bus module.

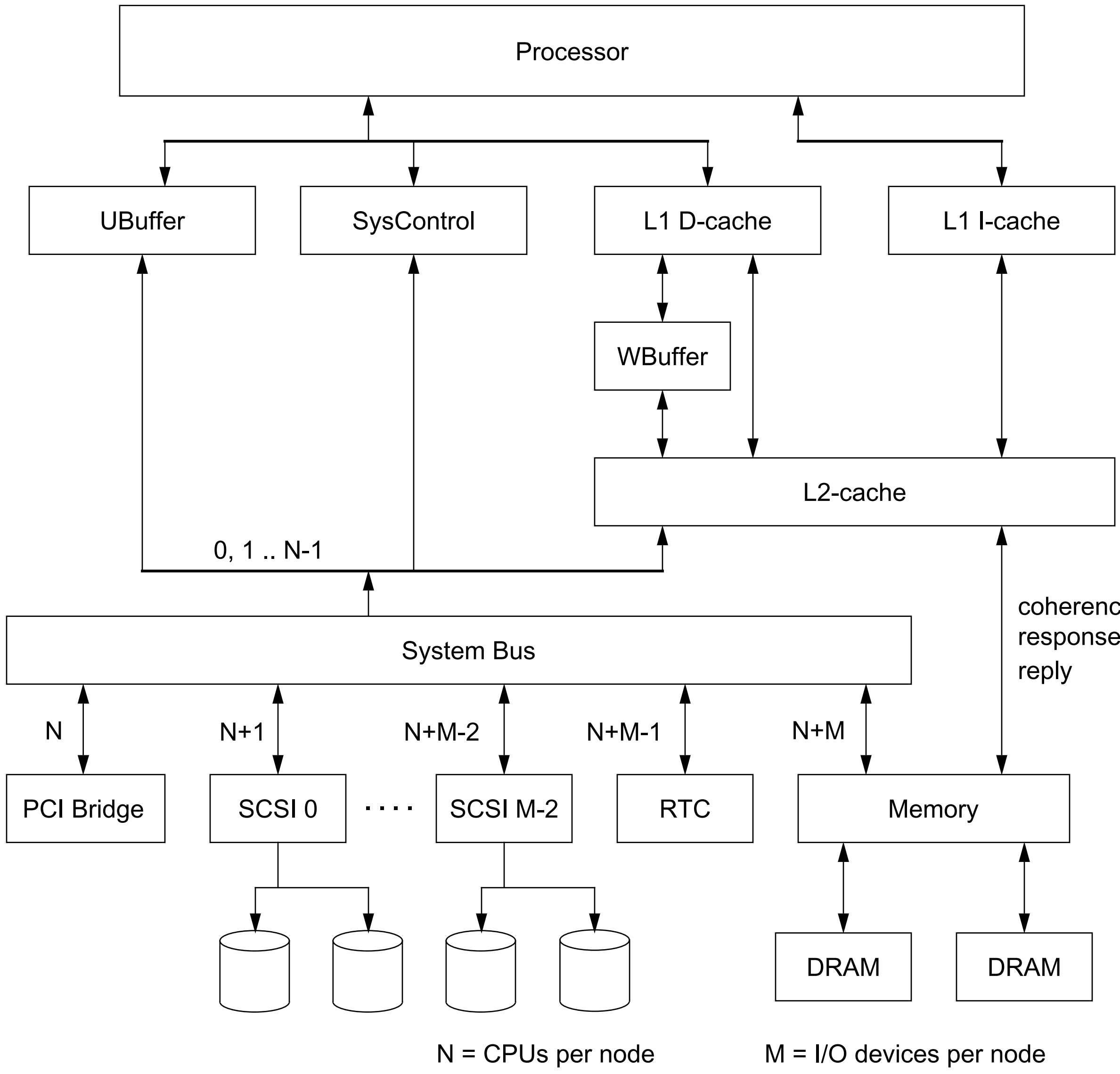

Figure 3: Node Architecture

1.1 Request Structure

files: Caches/system.c, Caches/names.c
Cachess/req.h, Processor/tlb.h

All levels of the memory hierarchy, including the system bus and I/O devices, operate on requests as the fundamental data structure. A request is described by the following fields:

- type (request, reply, coherency request, coherency reply, writeback ..)
- request number; used by bus module
- source node and bus module, destination node and bus module
- processor request type (read, write, rmw)
- request type (read shared, read own, read uncached ...)
- coherence type (shared, invalidate)
- virtual and physical address and address attributes
- request size in bytes
- flags indicating data is ready, coherency checks completed
- pointers to associated cache-to-cache copies or writebacks
- pointers to L1 and L2 MSHR entries
- request-specific data (instance and processor-ID, buffer pointer ...)

The following table summarizes the supported request types:

Type	Description
Read	bus module intends to read
Write	bus module intends to write
Rmw	bus module intends to perform atomic read-modify-write
FlushC	processor request to flush cache
PurgeC	processor request to purge cache
RWrite	remote write - unused
WBack	cache write back
Read_Sh	read shared (in response to a read request)
Read_Own	read exclusive (in response to a write request)
Read_Current	coherent read without cache line state change
Upgrade	change cache line state from shared to exclusive
Read_UC	read from uncached address space
Write_UC	write to uncached address space
Reply_Excl	data reply in exclusive state

Table 8: Request and Response Types

Type	Description
Reply_Sh	data reply in shared state
Reply_Upgrade	reply to upgrade request (no data transfer involved)
Reply_UC	data reply to read uncached
Invalidate	coherency response (implies cache-to-cache copy)
Shared	coherency response

Table 8: Request and Response Types

The first group of requests is used to identify the intention of the requesting bus module, it gets converted into the appropriate bus request type by the caches or bus interface module. The second group of requests is issued on the bus, it corresponds to the transaction types found in most MESI protocols. The request type *read_current* is used by I/O devices to read coherent data without transfer of ownership. Third-party caches that detect a dirty hit for such a request respond with a cache-to-cache transfer, but do not change the cache line state to *shared* or *invalid*. The request type *write_purge* is issued by I/O devices when writing an entire cache block of data. Third-party caches mark any blocks that match the request as *invalid*, without prior writeback of dirty data.

The third group of request types corresponds to response transactions. *Reply_excl* and *Reply_sh* involve data transfers and indicate the cache line status in the receiving cache. *Reply_upgrade* does not involve a data transfer, it is used to acknowledge an upgrade request after all coherency checks have been performed.

The two coherency response types do not involve bus transactions. Snoop responses are sent to the memory controller on separate dedicated channels.

Furthermore, the request data structure includes an attribute field that carries the page attributes from the TLB lookup. Encoding of the attribute field is the same as in the page table and TLB entries, and it should be accessed using the macros defined in tlb.h. The attribute field is set in *Cache_recv_addr* based on a similar field in the instruction instance object, and is used to distinguish normal memory accesses from uncached accesses. Furthermore, the uncached buffer (described later), uses the attribute field to distinguish normal and accelerated (combining) uncached accesses.

Instead of calling a fixed function upon completion of a request, each request contains two pointers to routines to be called when the request completes. This allows a variety of bus masters such as I/O devices to perform transactions without having to agree on the activity performed upon request completion. For instance, the processor memory unit needs to perform a load instruction and insert it into the memory done heap, while an I/O device may copy a block of data into an internal buffer and start the next DMA request.

1.2 Node Initialization and Statistics

files: *Caches/system.c*
Caches/system.h

The memory hierarchy is initialized by the routine *SystemInit*. It calls the initialization routines for each module of the memory hierarchy. These routines usually read the relevant parameters from the parameter file, allocate data structures and initialize the module. Note that these routines are responsible for initializing all instances of the respective module, for all nodes.

The routine *StatReportAll* is called upon completion of the simulation, or when requested by a simulator trap. For each module, it first calls a routine that prints the configuration of that module, and then a routine to print statistics information.

The routine *StatClearAll* is called when the statistics of all modules should be cleared. This can be done through a simulator trap. For each module, it calls a routine that clears the statistics for that module.

2 Caches

2.1 Cache Module Structure

files: Caches/cache.c, Caches/cache_help.c
Caches/cache.h

The figure below shows the components of the cache models. Each cache consists of three pipelines, a tag array, an optional data array, and a list of MSHRs (miss status holding registers). The request pipeline takes requests from the module *above*, that is the processor in case of an L1 cache and the L1 cache in case of the L2 cache. The reply pipeline receives replies from the lower levels, L2 cache or bus respectively. The coherency pipeline handles coherency requests, either due to coherent bus transaction from other bus masters (snooping), or due to subset enforcement at the L2 cache.

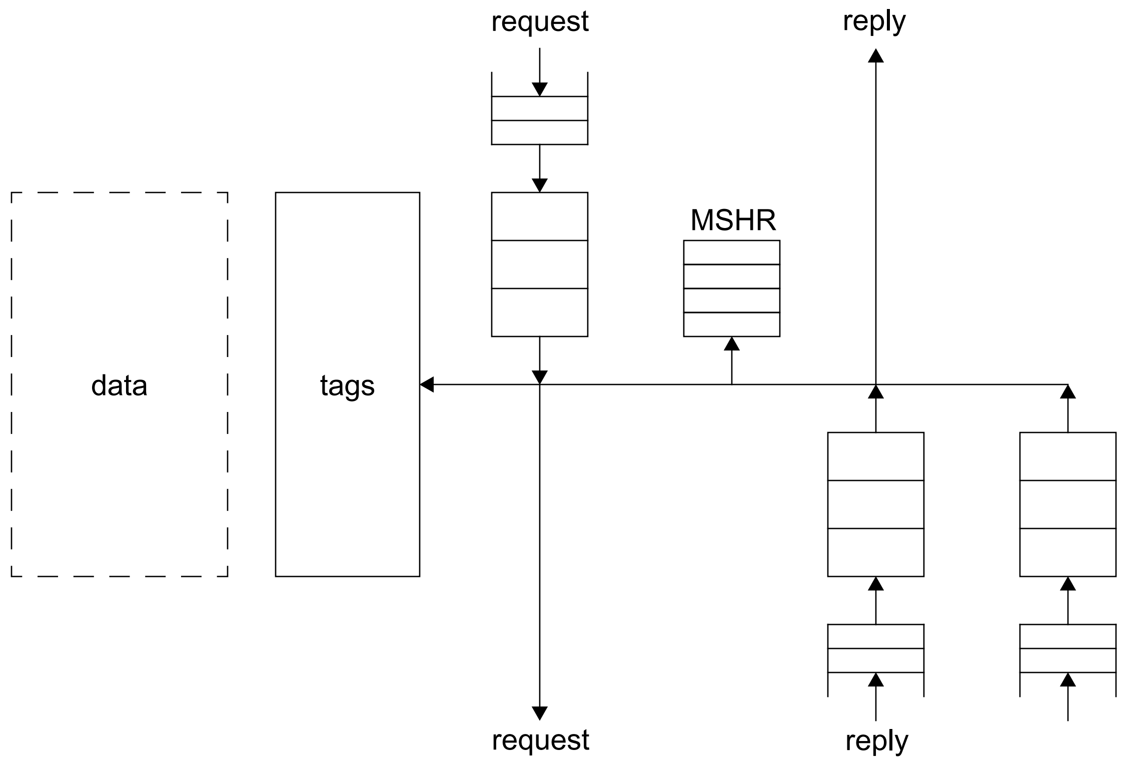

Figure 4: General Cache Architecture

The pipelines are used to model the various access latencies of the tag and data portions of the cache. Each pipeline has an associated input queue to buffer requests. In many cases, the queue is only used as a staging unit between modules within the same cycle. For instance, the processor places all memory requests that are issued in a cycle (limited by the number of memory units) in the L1 request queue, from where the L1 cache moves them into the request pipeline in the same cycle.

The MSHRs maintain a list of pending cache misses. When a new request arrives at the cache and it encounters a miss, the cache first checks the MSHR if the request can be merged with an already pending request, or if a new MSHR has to be (and can be) allocated.

The caches do not actually store any data, with the exception of the instruction cache. Physical memory is maintained in the nodes page table, which stores translations from simulated physical addresses to host memory, as well as pointers to the actual host memory. The cache models control when a memory access can be performed, they do not actually move data. The instruction cache differs from the other caches in that it stores pre-decoded instructions which are then fetched by the processor model.

Cache simulation is split into two phases. First, the cache handles all requests that produce a result. This includes processing data returns that unblock instructions in the processor, issuing requests to other cache modules, or starting arbitration for the system bus. This phase is simulated before the processor model simulation. When all processor models are simulated, the second cache phase is simulated. This phase consumes requests, for instance it removes requests from the 'virtual' cache port queues and inserts them into the appropriate pipelines. This arrangement allows the system to produce data from the caches, make it available to the processor and consume new request at the same cycle.

2.2 Cache Initialization

files: *Caches/cache_init.c*
Caches/cache_param.h

The cache initialization routine *Cache_init* reads all cache related parameters and creates the data structures for the L1 instruction and data cache and the L2 cache. For each cache, it allocates an array of cache structures and an array of pointers to these structures. Furthermore, for the L1 caches and the uncached buffer it allocates arrays of flags that indicate if the request queues of the respective modules are full. The routine then allocates a global statistics structure for each node. Finally, the routine calls the initialization routine for each cache instance it created, and initializes the write buffer.

The following table summarizes the configuration parameters for the cache hierarchy.

Parameter	Description	Default
Cache_frequency	cache frequency to processor frequency ratio	1
Cache_mshr_coal	maximum number of requests coalesced into one MSHR	8
L1IC_size	L1 instruction cache size in kbytes	32
L1IC_line_size	L1 instruction cache line size in bytes	32
L1IC_assoc	L1 instruction cache associativity	1 (direct-mapped)
L1IC_perfect	L1 instruction cache has perfect hit ratio (not supported)	0 (off)

Table 9: Cache Parameters

Parameter	Description	Default
L1IC_ports	number of CPU ports for L1 instruction cache	2
L1IC_prefetch	enable prefetching upon cache miss	0 (off)
L1IC_mshr	number of MSHRs for L1 instruction cache	8
L1IC_tag_latency	L1 instruction cache latency in cycles	1
L1IC_tag_repeat	L1 instruction cache repeat rate	1
L1DC_size	L1 data cache size in kbytes	32
L1DC_line_size	L1 data cache line size in bytes	32
L1DC_assoc	L1 data cache associativity	1 (direct-mapped)
L1DC_writeback	enable writeback mode for L1 data cache	1 (on)
L1DC_perfect	L1 data cache has perfect hit ratio	0 (off)
L1DC_ports	number of CPU ports for L1 data cache	2
L1DC_prefetch	enable prefetching upon cache miss	0 (off)
L1DC_mshr	number of MSHRs for L1 data cache	8
L1DC_tag_latency	L1 data cache latency in cycles	1
L1DC_tag_repeat	L1 data cache repeat rate	1
L2C_size	L2 cache size in kbytes	256
L2C_line_size	L2 cache line size	128
L2C_assoc	L2 cache associativity	4
L2C_perfect	L2 cache has perfect hit ratio	0 (off)
L2C_ports	number of request ports from L1 caches	1
L2C_prefetch	enable prefetch upon cache miss	0 (off)
L2C_mshr	number of MSHRs for L2 cache	8
L2C_tag_latency	L2 cache tag access latency in cycles	3
L2C_tag_repeat	L2 cache tag access repeat rate	1
L2C_data_latency	L2 cache data access latency	5
L2C_data_repeat	L2 cache data access repeat rate	1

Table 9: Cache Parameters

Each cache initialization routine initializes various fields in the respective cache structure, such as node and processor ID, cache configuration and replacement policy. It then allocates and initializes the miss status holding registers (MSHR), the input queues and processing pipelines, and the per-cache statistics structures. The routine *Cache_init_aux* allocates the tag array for a cache and initializes every tag.

2.3 Cache Statistics

Statistics is collected on a per-request basis. As a request passes through the cache hierarchy, its issue time and hit type is recorded in the request structure. When a CPU-issued request completes in the routine *Cache_global_perform*, the routine *Cache_stat_set* is called to update the cache statistics based on the information recorded in the request. Each cache hierarchy gathers statistics about the number of requests broken down by type as well as a total, the number of hits for each level of cache and the average latency of requests.

2.4 L1 Instruction Cache

2.4.1 Processor Interface

files: Caches/cache_cpu.c, Processor/memprocess.cc

Instruction requests are issued to the cache during the instruction fetch stage. The processor model calls the routine *ICache_recv_addr* with the virtual and physical address and the number of instructions to fetch as arguments. The routine allocates a request structure and fills in the relevant elements. A queue is used to logically stage the requests before they are handled by the cache simulation routine. Requests are inserted into the queue when the processor model simulates, and are removed during the same cycle when the caches are simulated. In the case of the instruction cache, at most one element is ever in the queue.

Instruction fetches are completed in the routine *PerformIFetch*, which is called when the request itself completes. For every open fetch queue slot, up to the maximum number of instructions fetched per cycle, the routine allocates an instance structure, copies the static pre-decoded instruction from the cache data array into the instance and loads the instance into the fetch queue.

2.4.2 Handling Incoming Requests

files: Caches/l1i_cache.c

The routine *L1ICacheInSim* is called by the main simulation event when the input queue is not empty. It is responsible for the three input queues of the cache. The processor request queue has as many entries as the processor issues request in each cycle. It operates as a buffer between the processor and the cache within the same cycle. Requests are put in the queue by the processor model only if the queue is not full, and are removed from the queue in the same cycle when they enter the processing pipeline.

The input routine first checks if the reply queue is not empty, removes a request if this is the case, and places it into the request pipeline if possible. The reply queue is processed first because these requests are holding resources, such as an MSHR entry, that may be released when processing the reply. Next, the routine checks for requests in the request queue and moves those to the request pipeline, and finally it does the same for the coherency queue, which is used for invalidation messages due to coherency traffic or subset enforcement at the L2 cache.

If a perfect instruction cache is simulated, all incoming requests are treated as cache hits and complete immediately. In this case, the respective completion routines are called when the request is received, to load the requested instructions in their pre-decoded form into the cache data array.

2.4.3 Handling Outgoing Requests

files: Caches/l1i_cache.c, Processor/memprocess.cc

Outgoing requests are handled in each cycle before the processor is simulated. The routine *L1ICacheOutSim* is called by the main simulation event when at least one request is in one of the cache pipelines, as indicated by a flag. It first checks the reply pipeline for request, then the request pipeline, and finally the coherency pipeline. Each type of request is handled by a different routine.

Replies are handled by the routine *L1IProcessTagReply*. This routine first updates the statistics fields of the L1 instruction cache and of this particular request. It then calls the routine *PredecodeBlock* which pre-decodes the instructions in this cache line and writes them into the cache data array. Pre-decoded instructions are essentially the static portion of an instance structure. Finally, the request and all requests that have been coalesced into the same MSHR entry can be completed, and the MSHR entry is released. Completing a request involves calling the *perform* routine whose pointer is provided in the request structure, and returning the request to the request pool.

Instruction fetches are performed by the routine *PerformIFetch*. It first checks if the expected fetch PC is equal to the virtual address of the reply, and if the current processor status word (*pstate*) is equal to the *pstate* value at the time the instruction fetch was started. This allows the fetch unit to detect cases when the processor mode changed (e.g. disabled the I-TLB) while the instruction fetch was in flight or the decode stage has redirected instruction fetches. If both *pc* and *pstate* have the expected value, it allocates an instance structure for every instruction in the reply and copies the static pre-decoded portion from the I-Cache data array into the instance structure, and inserts the instance into the fetch queue. It continues this until either all requested instructions have been fetched, or the fetch queue is full.

Processor requests are processed by the routine *L1IProcessTagRequest*. First, this routine checks if a MSHR entry needs to and can be allocated by calling *L1ICache_check_mshr*. If the request matches an existing MSHR entry and the entry has space for another request, the new request is coalesced in the MSHR entry, otherwise the request stalls. If no matching MSHR entry is found, and the request is a cache miss, a new MSHR entry is allocated if one is available, otherwise the request is stalled. If the request is a cache hit, no MSHR entry is allocated. Depending on the return value of this routine, the request handling routine performs the request immediately (in case of a hit), drops the request if it has been coalesced, or adds it to the L2 request queue. Optionally, upon a cache miss, a sequential prefetch request may be issued to the L2 cache.

Coherency requests are caused by external coherency traffic that hits an L2 cache line, or by subset enforcement in the L2 cache. These requests are handled by *L1IProcessTagCohe*. If the request hits in the instruction cache, the corresponding cache line is invalidated, and the request is returned to the request pool.

2.5 L1 Data Cache

2.5.1 Processor Interface

files: Caches/cache_cpu.c, Processor/memprocess.cc

Data requests are issued to the data cache from the memory unit when the instruction is ready to issue, either after address calculation for loads, or when a store or uncached operation is about to graduate. To issue a request, the processor model calls *DCache_recv_addr* with the virtual and physical address and request size as arguments. The routine allocates a request structure, fills in the relevant elements and inserts it into the cache request queue. Similarly to the instruction cache, the queue is only used to stage requests during the same cycle. The number of requests is limited by the number of memory requests that the processor can issue in a cycle.

Data cache requests are complete by two routines. *PerformData* executes the instruction that corresponds to the request, i.e. it performs the load or store. In addition, instructions must be marked as complete by inserting them into the done-heap (routine *MemDoneHeapInsert*), from where they will be removed later while dependent instructions are released.

2.5.2 Handling Incoming Requests

files: Caches/l1d_cache.c

Incoming requests are handled by the routine *L1DCacheInSim*. Similarly to the instruction cache, this routine checks the reply queue, request queue and coherency queue for waiting requests and inserts them into the corresponding cache pipelines. This routine is also responsible for detecting requests to the local system control page (described in a later section) and forwarding them to the system control module. In addition, uncached requests are sent to the uncached buffer instead of being handled by the cache.

2.5.3 Handling Outgoing Requests

files: Caches/l1d_cache.c, Processor/memprocess.cc

This stage is again similar to the instruction cache, with the main difference being that cache misses may lead to write back requests which are issued to the L2 cache or stalled in the L1 cache if the L2 cache is busy. The general flow of requests is the same as in the instruction cache, so it is not discussed here.

2.6 L2 Cache

2.6.1 Handling Incoming Requests

files: Caches/l2_cache.c

The L2 cache differs from the L1 caches in that it implements the bus interface, deals with cache-to-cache copies and forwards coherency message to the L1 caches. In addition, it implements two separate pipelines, one for tag accesses and one for data accesses. Requests always first access the tags, and then data if necessary. Requests arrive either from the L1 caches, or from the bus interface in the form of snoop requests or replies to previous requests.

The routine *L2CacheInSim* handles all requests that arrive at the cache. It first checks if there is a pending cache-to-cache copy from a previous cycles and adds it to the data pipeline. It then removes a request from the coherency queue and inserts it into the tag pipeline, since coherency requests must be snooped. Noncoherent requests may arrive at the cache if they are targeted at the external system control page, in which case they are handled by the system control module. Next, replies are inserted into the reply pipeline, which is logically similar to the data pipeline since replies usually contain data. Finally, the routine handles new requests from an L1 cache and also inserts them into the tag pipeline. Notice that the order in which requests are handled implies a priority, i.e. a coherency request may prevent a L1 request from progressing if the tag pipeline stalls.

2.6.2 Handling Outgoing Requests

files: Caches/l2_cache.c, Caches/cache_bus.c

Similarly to the L1 caches, the routine *L2CacheOutSim* removes requests from the various pipelines and performs most of the actual work. It first removes requests from the data pipeline and determines the appropriate action. If the request is an L1 request and it hits in the L2 cache, it is returned to the L1 cache. If it is a writeback and it hits in the cache it is complete at this point and can be dropped. If the request is a reply in response to an L2 miss, the cache releases all requests that coalesced in this cache line and sends the reply up to the L1 cache. If a cache-to-cache copy or writeback arrives at the data array, it is send out on the bus.

Requests at the tag array are either moved into the data pipeline if they require a data access (such as a hit in the cache), merge with existing MSHR entries or allocate a new entry and are sent out on the bus.

External coherency requests are dropped if they are from the same L2 cache, otherwise the cache line state is updated according to its current state and the request type. In addition, the L2 cache needs to invalidate the L1 cache if either a snoop request or an L2 cache miss replace a cache line. If the line is replaced because of a request from an L1 cache, the invalidation request is only send to the other cache, otherwise both caches receive it. If a dirty cache line is hit by a coherency request, it needs to be sent out on the bus as a cache-to-cache copy. It is also possible that an earlier writeback that is waiting in the outgoing buffer matches a coherency request, in which case it must be converted into a cache-to-cache copy.

2.6.3 Bus Interface

files: Caches/cache_bus.c

The bus interface is part of the L2 cache. It consists of a request buffer and a queue where requests wait for their turn to arbitrate. The L2 cache sends requests to the bus interface by calling *Cache_start_send_to_bus*. This routine sets the *issue_time* and *bus_start_time* elements of the request to the current time and sets the correct number of bus cycles in the request structure. Normal cached requests also allocates an entry in the request buffer, while writeback requests and replies (cache-to-cache copy) are queued in the outbuffer. If the request has been inserted successfully into the correct queue or buffer, the routine either starts arbitrating for the bus if the cache currently does not own the bus and the request is the first request to arrive at the system interface. Otherwise, the request is inserted into the list of arbitration waiters, from where it will be removed when the request ahead of it issues on the bus.

Once the cache wins arbitration, or if it has been the bus owner and a new request arrives, the bus calls the routine *Cache_in_master_state*. This routine schedules the bus events to deliver the request and complete it (possibly at the same time if critical-word-first is not enabled), removes the next request from the *arbwaiters* queue and arbitrates again for the bus.

When the request is delivered to the target module, the bus calls *Cache_send_on_bus*, which depending on the request type calls a bus module routine to actually send the request, and removes writebacks and replies from the outbuffer.

3 System Control Module

The system control module contains configuration registers that describe various system parameters, such as number of CPUs, cache and TLB size and clock period. It can only be accessed using uncached reads and writes. Each system control module provides one page of storage, and each CPU has its own copy. The module can be addressed in two distinct ways.

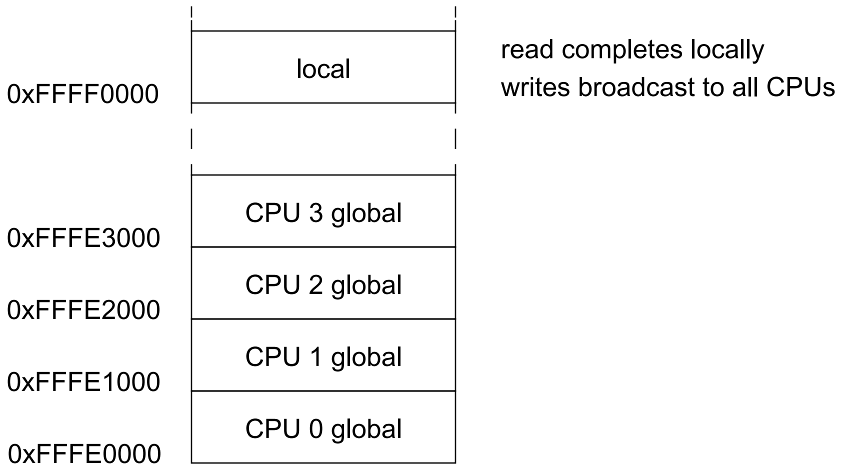

Figure 5: System Control Page Layout

Read accesses to the local address space are satisfied locally, no bus transaction is generated. Writes to this address space are broadcast to all CPUs. This can be used to broadcast interrupts. Accesses to the private (global) are directed at a specific CPU only, they always result in an uncached bus transaction.

3.1 Address Map

files: Caches/syscontrol.h

The following table lists the fields that are defined within the system control address region.

Offset	Description
0x00	CPU number within node
0x04	node ID
0x08	total number of CPUs in node
0x0C	total number of nodes
0x20	L1 instruction cache size
0x24	L1 instruction cache block size
0x28	L1 instruction cache associativity

Table 10: System Control Module Address Map

Offset	Description
0x30	L1 data cache size
0x34	L1 data cache block size
0x38	L1 data cache associativity
0x40	L2 cache size
0x44	L2 block size
0x48	L2 associativity
0x50	Instruction TLB type (fully assoc., set assoc., direct mapped)
0x54	Instruction TLB size
0x58	Data TLB type
0x5C	Data TLB size
0x60	clock period
0x64	physical memory size
0x80	external interrupt

Table 10: System Control Module Address Map

Addresses up to offset 0x80 are read only, the system control module prints an error message when it receives a write to such an address. Writing to the interrupt location triggers an external interrupt. The lower 8 bits that where written into this location are used as interrupt vector. The interrupt register can be written by the CPU itself as well as by external bus masters.

3.2 System Module Initialization

files: Caches/syscontrol.c, Caches/system.c
Caches/syscontrol.h

The system control module initialization routine *SysControl_init* is called from *SystemInit*. The routine first inserts one entry for the local system control page in each node. This address map entry has no corresponding physical memory, accesses to this region are redirected to the distinct global pages. The target CPU specified in this address map entry is a special target *ALL_CPUS*, which indicates to the bus to broadcast requests in this address range to all CPU modules. The initialization routine then inserts address mappings and page table entries for each private page, and initializes the read-only locations with the appropriate system parameters.

3.3 Handling CPU Requests

files: Caches/syscontrol.c, Caches/l1cache.c, Caches/l2cache.c, Bus/bus.c
Caches/syscontrol.h

The L1 cache (or if a single-level cache hierarchy is simulated the L2 cache) removes entries from the cache-port queue and forwards them to the uncached buffer, the system control module or the cache. If the request is a read from the local address range, it is handed to the routine *SysControl_local_request*(), which completes the read by changing the request and instance addresses to the correct private address, performing the respective memory operation and inserting the instruction into the processors memory done heap.

A write to the local address range as well as reads and writes from the private system control regions are forwarded to the uncached buffer, which will issue a bus transaction. Since the target module for requests in the local system control region is *ALL_CPUS*, the bus broadcasts such requests by duplicating the request and sending it to all processors.

3.4 Handling External Requests

files: Caches/syscontrol.c, Caches/cache_bus.c
Caches/syscontrol.h

The system interface (located in the L2 cache), implements a queue for incoming non-coherent requests. The routine *Cache_get_noncoh_request* is called by the bus module when a processor receives a non-coherent (usually I/O) request, it inserts the request in the target CPUs non-coherent queue. When the L2 cache takes requests out of the incoming queues (coherent and non-coherent), it checks if the request address is in the global or local system control address region. If this is the case, the request is handled by the routine *SysControl_external_request*, otherwise it is added to the normal cache request pipeline.

The system control module first checks if the request is valid, that is if it is either a read or write uncached, and if it is not a write to a read-only register. If the request address is in the local region, it is changed to the correct global region. For write requests, the routine then performs each of the possibly coalesced requests and returns the request structure to the global pool. If any of the requests is a write to the interrupt register, the routine *ExternalInterrupt* is called to signal an interrupt to the processor. For reads, the module performs each of the coalesced reads and then issues a response by calling *Cache_start_send_to_bus*.

4 Uncached Buffer

Figure 2 in section 4 shows the memory hierarchy in ML-RSIM. In addition to a conventional cache hierarchy it includes an uncached buffer that manages uncached I/O space memory operations that bypass the caches. Uncached requests are demultiplexed by the L1 cache module and sent to the uncached buffer. The system interface multiplexes requests from the caches and the uncached buffer before sending them to the system bus.

Load and store instructions to I/O space are handled by the uncached buffer, after they have been issued to the memory hierarchy. The processor issues these instructions non-speculatively and strictly in-order. The uncached buffer stores and possibly combines these memory accesses before issuing them to the system bus.

4.1 Initialization

files: Caches/system.c, Caches/ubuf.c
Caches/ubuf.h

Various aspects of the uncached buffer can be configured. The configuration parameters are summarized in the following table:

Parameter	Description	Default
ubuftype (combine/nocombine)	coalesce requests	combine
ubufentrysize	number of bytes per entry	8
ubufsize	number of entries	4
ubufflush	threshold for flushing buffer	1

Table 11: Uncached Buffer Configuration Parameters

The parameter *ubufsize* specifies the number of entries (default 4), *ubufentrysize* specifies the width of each entry in bytes. This parameter limits the number of requests that will be combined in one entry. The uncached buffer can be configured to combine memory references into a single entry (ubuftype *combine*, default), or to allocate a new entry for each request (*nocombine*).

The buffer flushing policy can in part be changed by varying *ubufflush*. This parameter specifies the minimum number of entries that needs to be occupied before the buffer is flushed. Note that flushing is also initiated when a read request or a memory barrier is present in the buffer.

Like any other part of the memory hierarchy, the uncached buffer is instantiated and initialized in the *System_init* routine. This routine creates one uncached buffer with the specified parameters for each node and connects it with the cache module that demultiplexes the requests and the system interface. The initialization routine *UBuffer_init* creates various data structures, copies the configuration parameters and clears the statistics variables.

4.2 Issuing Uncached Memory Operations

files: Processor/memunit.cc, Caches/ubuf.c
Caches/cache.h, Caches/ubuf.h

Uncached memory operations are issued to the memory hierarchy non-speculatively and in-order by the processor. Requests from the processor are buffered in a queue before being handled by either the cache or the uncached buffer. Note that this queue is merely a simulator abstraction, it does not model actual cache behavior. The number of transactions that can be issued per cycle is limited by the number of load/store units that are configured, these units are marked as busy until the request has been removed from the queue. The only purpose of the queue is to buffer all requests that are issued in the same cycle.

The request queue is drained by the L1 cache module in the *L1CacheInSim* routine. It examines the memory attributes of each incoming request, if the request targets uncached address space it calls *UBuffer_add_request*, otherwise it inserts the request into the cache pipeline.

4.3 Uncached Buffer Structure

files: Caches/ubuf.c
Caches/ubuf.h

The uncached buffer is represented by a data structure that contains a counter that indicates how many entries are currently used and an array of pointers to the individual entries. Each entry consists of an array of requests (up to a cache line worth of 32 bit transfers), the cache line aligned address (used for combining), fence and busy bits and the access type (read or write).

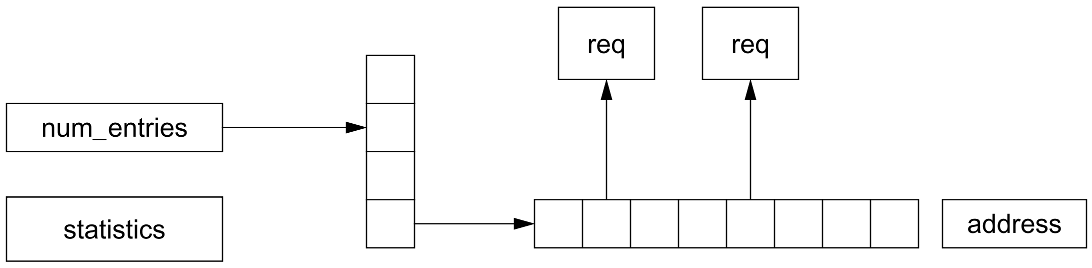

Figure 6: Uncached Buffer Structure

Initially, no entry is valid in the buffer. Whenever a new request cannot combined with an existing entry, a new entry is allocated and the request inserted in the request array based on its offset with the cache line.

The uncached buffer also maintains pointers to the cache module that contains the request queue, which is usually the L1 cache, and a pointer to the cache module that contains the system interface. The first pointer (*above*) is used to remove an entry from the request queue if it was processes successfully, while the second pointer (*below*) is used to pass requests to the system interface.

To synchronize the uncached buffer with the system interface, the uncached buffer maintains a flag *transaction_pending*, which if set indicates that a transaction has been sent to the system interface but not yet processed. The system interface clears this flag when a write transaction is ready to issue to the bus, or when a read transaction has received the data reply. The uncached buffer will not issue further transactions until the pending transactions has been processed by the system interface. This scheme ensures that uncached transactions are issued strictly in-order, and also models the necessary flow control between uncached buffer and system interface.

4.4 Handling Incoming Requests

files: Processor/procstate.cc, Caches/ubuf.c

The routine *UBuffer_add_request* is called by the L1 cache module when it received an uncached request. The routine is expected to remove the entry from the request queue and return 1 if the request has been processed successfully, otherwise it returns 0.

If the incoming request is a barrier and the uncached buffer is not empty, the fence flag will be set in the youngest entry, thus preventing combining of requests beyond this entry. Also, to initiate flushing the uncached buffer, the read-count variable is incremented. The barrier can be ignored when the buffer is empty.

The other two possible requests are reads and writes. Initially, the index where a new request will be inserted is set to the top of the buffer. If combining is enabled, the routine searches existing entries towards the bottom of the buffer until it finds a suitable entry. A suitable entry is found when the request address is within the same cache line, the request is of the same type and the buffer entry is not busy. The search is aborted when an entry is marked as a fence or when the types don't match, thus preventing that reads and writes bypass each other or bypass a fence. A sequence of only reads or writes, however, may be reordered if subsequent accesses are to different cache lines. Combining is possible only for 4 or 8 byte accesses, smaller requests will always allocate a new entry. Furthermore, it is possible that later requests overwrite older requests to the same address.

If combining is not possible or disabled and the buffer is not full, a new entry will be allocated at the top. It is initialized with the request type (read or write) and cache line aligned address for later combining.

At this point, regardless of whether the current request will be combined with an existing entry or allocated a new entry, the request is placed in the slot corresponding to the offset within the cache line. For example, a store to address 0xFFF00108 will be inserted in slot 2.

The number of entries that is potentially combined can be varied by means of the parameter *ubufentrysize*. This specifies the number of bytes in each entry, it is the sum of transfer sizes of all distinct requests. If the request just inserted is a read the read-count variable will be incremented to initiate flushing the buffer.

After the current request has been processed, it is removed from the request queue located at the L1 cache and the entry-counter is incremented.

4.5 Sending Requests to the System Bus

files: *Caches/ubuf.c*
Caches/ubuf.h

RSIM_EVENT calls *UBuffer_out_sim* before simulating the processor (at the beginning of the cycle) if the uncached buffer is not empty, or the conditional store buffer contains a waiting entry. This routine first looks for the first valid request within the oldest buffer entry by searching through the slots beginning at 0. When such a request is found, it will be sent to the system bus. It may, however, be possible to combine it with other requests that might be present in the same entry.

The function *MaxCoalesce* returns the number of requests that can be combined, based on the request size and address alignment. A bus transaction must always be naturally aligned based on its transaction size. For instance, a 4-byte store to address 0xFFF00108 maybe combined with at most one more 4 byte request at address 0xFFF0010C.

This function first scans the entry slots to find the maximum number of consecutive requests, taking into account the request sizes. This number is than rounded down to the largest power of two that is equal to or smaller than the address alignment. The simulation function then processes the following N-1 requests (where N is the number of requests that can be coalesced). Requests are coalesced by forming a linked list of requests. If a read request has been sent to the bus or an entry marked as fence has been processed completely, the read-count variable is decremented, which might prevent further flushing of the buffer. Note that this method of combining may result in more than one bus transaction even for requests within the same buffer entry. In this case, the busy bit will be set for the oldest entry, thus preventing new requests from being inserted in this entry.

Before sending a request to the system interface, the uncached buffer sets the flag *transaction_pending*. This flag serves as flow control and ensures that uncached requests are issued on the bus in order. Requests are sent to the system bus by calling *Cache_start_send_to_bus*. This routine inserts the request in a queue of waiting requests, from where request will later be issued to the bus. The system interface notifies the uncached buffer that a transaction is ready to issue (write), or that the request is completed (read) by calling *UBuffer_confirm_transaction*, which may cause the uncached buffer to send the next request to the system interface

4.6 Statistics

files: *Caches/system.c, Caches/ubuf.c*
Caches/ubuf.h

During simulation, the uncached buffer collects various statistics information like number of stall cycles, maximum size of coalesced requests and absolute number of coalesced requests. In addition, the buffer utilization is sampled and will be displayed as a histogram.

It also counts the total number of memory barriers that have been received and the number of effective barriers, that is the number of barriers that have been marked in an existing buffer entry.

5 System Bus

5.1 Bus Configuration

files: Bus/bus.c
Bus/bus.h

The system bus models a multiplexed split-transaction bus with or without critical-word first data transfers. Configuration parameters include the arbitration delay, turnaround cycles and the minimum delay between consecutive data transfers. The following figure shows the relationship between these values.

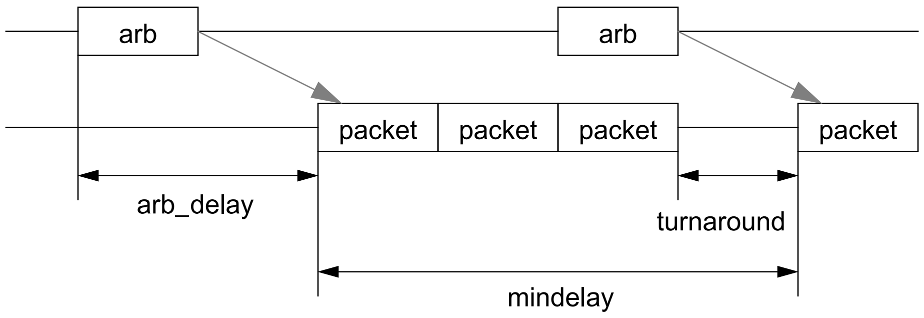

Figure 7: Pipelined Bus Model

The table summarizes all configuration parameters.

Parameter	Description	Default
bus_width	width of data path in bytes	8
bus_frequency	frequency ratio to CPU core	1
bus_arbdelay	arbitration delay	1
bus_turnaround	number of turnaround cycles	1
bus_mindelay	minimum delay between transactions	0
bus_total_requests	maximum number of outstanding coherent requests	8
bus_cpu_requests	per-processor number of coherent requests	4
bus_io_requests	per-I/O device number of coherent requests	4
bus_critical_word	enable or disable critical word first data returns	1 (on)

Table 12: Bus Configuration Parameters

The arbitration delay parameter specifies the latency between arbitration and start of packet transfer. Realistic values would be 1 or 2 cycles. A bus turnaround cycle is necessary in most systems whenever the device that is driving the bus changes.

The parameter *bus_mindelay* is intended to model a bus with reactive flow-control, where slaves acknowledge each transaction (the address portion). If the system interface insures strong ordering, it cannot issue the next transaction before the previous one has been acknowledged. This parameter affects only transactions that are shorter than the specified minimum delay, it effectively inserts a number of idle cycles after such transactions.

Each split transaction (read) requires a unique transaction ID. The parameter *bus_total_requests* sets the maximum number of outstanding transaction on a bus, while the *bus_cpu_requests* and *bus_io_requests* parameter limit the number of outstanding requests per processor or I/O device. The *bus_critical_word* parameter specifies when the response is sent to the requestor (cache, I/O module). If the parameter is set to 1, the bus module delivers the response to the module 1 bus cycle after the transaction has been issued on the bus. This approximates a critical-word-first data return, since the data return is consumed by the cache as soon as the first word has been transferred. However, if other cache misses have been coalesced into the request, they too are completed after the bus cycle.

5.2 Bus Transactions

files: Bus/bus.c, Caches/cache_bus.c, Caches/l1_cache.c, Caches/l2_cache.c, IO/io_generic.c Caches/req.h/

The system bus supports coherent and non-coherent bus transactions to implement the MESI coherency protocol, as well as a variety of transactions for I/O device access and optimized I/O device DMA transfers., The following table summarizes the supported transaction types and gives a short description of each.

Transaction	Coherent	Description
READ_SH	yes	get cache block for CPU read, third-party state changes to shared, data may be returned shared or exclusive from main memory of third-party cache
READ_OWN	yes	get cache block for CPU write, third-party invalidates upon a hit, data may be returned from main memory of third-party cache
READ_CURRENT	(yes)	get cache block for I/O device DMA read, third-party cache block state does not change, data may be returned from main memory or third-party cache
UPGRADE	yes	change cache block state from shared to exclusive, no data transfer
READ_UC	no	read cache or sub-block from I/O device
WRITE_UC	no	write cache or sub-block to I/O device
REPLY_EXCL	no	return data in exclusive mode from main memory or third-party cache
REPLY_SH	no	return data in shared mode from main memory or third-party cache
REPLY_UPGRADE	no	confirms upgrade request, no data transfer involved
REPLY_UC	no	return data in response to a read_uc request

Table 13: System Bus Transactions

Transaction	Coherent	Description
WRITEBACK	no	write cache block back to main memory
WRITE_PURGE	yes	write cache block to main memory and invalidate matching blocks in third-party caches (used for I/O device DMA write)

Table 13: System Bus Transactions

The first set of transactions (except for read_current) implements the basic MESI cache coherency protocol. *read_current* is an optimization for DMA reads. It is coherent in a sense that third-party caches snoop it and return the requested cache block if it is found in modified exclusive state, but the cache block status is not changed. This allows DMA devices to coherently read data without otherwise affecting the cache. Uncached requests transfer a variable amount of data, from 1 byte to an entire cache line, depending on the combining strategy used. *WRITE_PURGE* is another optimization to speed up DMA writes. The transaction writes an entire cache block to main memory while invalidating matching blocks in third-party caches. This avoids the need for the DMA device to acquire ownership of a cache block before writing it to main memory.

5.3 Trace File

files: Bus/bus.c, Processor/mainsim.cc
Bus/bus.h, Processor/simio.h

If the command line flag '-db' is set, the bus model writes a trace file that contains information about the arbitration phase and beginning/end of each transaction.

The trace file name and directory is formed similarly to the other simulator output files, based on the command line parameter -D and -S for directory and subject name, extended with '_bus.X' where X corresponds to the node number.

The trace file lists the time a transaction is started along with transaction type, size, number of cycles of the transaction and source and destination port within the bus module. The subsequent line usually indicates when the transaction was finished, which is also the time the result will be available to the processor.

5.4 Bus Operation

files: Bus/bus.c
Bus/bus.h

The bus model is completely event driven. Three different events are scheduled by bus masters or by the bus itself to initiate arbitration, issue transactions and finish transactions. The bus can be in any of three states: idle, arbitrating, or active. The bus maintains an array of pending requests, one per bus module. Bus modules are expected to provide additional buffering internally if necessary. In addition, the bus implements a write flow control counter and counters of pending total as well as CPU and I/O module requests, plus a round-robin pointer to a bus module.

To start a bus transaction, modules first check the bus status. If the bus is idle and the module currently owns the bus, it can immediately start the transaction. Otherwise, it notifies the bus of its intent by calling *Bus_recv_arb_req*. This routine inserts the request in the appropriate bus module's pending slot and schedules the arbitration routine to in the current cycle, or at the end of the current transaction, whichever is longer. In addition, this routine adjusts event times to be aligned with the bus frequency.

The arbitration routine *Bus_arbitrator* applies a round-robin strategy and searches for the next bus module with a pending transaction. If write flow control is enabled, write transactions are skipped during this process. Similarly, request transactions are skipped if the total number of pending transactions has been reached. The memory controller has always highest priority to avoid starvation of modules waiting for data responses. If no eligible transaction was found, the bus arbitrator event reschedules itself for the next bus cycle, essentially polling until the flow control state or pending transaction count changes. After deciding on the next transaction, the arbitration routine computes the cycle that the next transaction begins driving the bus. If the bus owner changes, it enforces the specified turnaround cycles. In addition, it ensures that the beginning of transactions is separated by at least *bus_min_delay* cycles. Based on these considerations, it schedules the routine *Bus_wakeup_winner* to occur at the first cycle of the next transaction.

Bus_wakeup_winner increments or decrements the pending request counter, depending on the request type, and calls an *in_master_state* routine for the cache, I/O module or memory controller. This routine notifies the bus module that its oldest transaction began driving the bus. The routine is expected to set a variable that indicates for how many cycles the bus is busy, and then call *Bus_start* to indicate that the bus started a transaction. This module-class specific routine is called immediately if the module determines it owns the bus and the bus is idle when it first attempts to start a transaction. If other requests are waiting and the bus arbitrator is not yet scheduled, the *Bus_wakeup_winner* routine then schedules the arbitrator to execute at the end of the current transaction, otherwise it marks the bus as idle.

The *Bus_start* routine updates the current bus owner pointer and schedules a notification event to either occur at the same cycle if critical-word first is enabled and the transaction is a data response, or at the end of the transaction, plus a completion event. The notification event is handled by the routine *Bus_perform*. This routine calls a module-class specific *send_on_bus* routine that notifies the source module that transaction can be considered arriving at the destination module. Remember that this event happens at the beginning of a data response transaction if critical-word first is enabled, assuming that the target module can start consuming the data at the first cycle. For other transactions and for data-response transactions without critical-word first, the event happens at the end of the transaction. The module-specific *send_on_bus* routine essentially calls a bus routine to forward the request structure to the target module, and it may also check if a writeback transaction has been converted into a cache-to-cache transfer because of a snoop hit while the request was waiting for the bus. The completion event (routine *Bus_issuer*) only updates some statistics variables.

The bus provides several different routines to actually transmit a request structure from the source to the target module. Coherent requests need to be broadcast among all coherent bus modules.

Bus_send_request forwards the request to all caches (*Cache_get_cohe_request*) and I/O modules (*IO_get_cohe_request*) and then sends the request to the memory controller (*MMC_recv_request*). Non-coherent but cached requests are only forwarded to the memory controller. Uncached requests (including uncached writes) are handled by *Bus_send_IO_request*. This routine determines whether an uncached request targets an I/O module or a processor system control module. In the first case, the routine calls *IO_get_request* to forward the request to the target module. If the request targets a system control module, it may be a broadcast write to all modules, in which case the routine sends individual copies to all processor bus interfaces. Otherwise, it sends the request to the target module. *Writepurge* transactions are handled by *Bus_send_writepurge*. Similarly to coherent requests, this routine sends the same request to all coherent modules for snooping, plus to the memory controller. *Writeback* transactions are always non-coherent and are only send to the memory controller (*Bus_send_writeback*). Data responses are issued only by the memory controller by calling *Bus_send_reply*. This routine forwards the reply either to a cache or an I/O module. Coherent data replies are sent by caches or coherent I/O devices as cache-to-cache transfers. The routine *Bus_send_cohe_data_response* forwards the request to the original request module, and sends a copy to the memory controller. Finally, I/O replies are always send to a cache, as it is currently assumed that I/O devices never issue uncached read requests. The appropriate bus function to forward a particular request to the target module is determined by the module-class specific *send_on_bus* routine.

5.5 Bus Statistics

The bus maintains a few statistics variables that count the total number of transactions and the number of cycles the bus was in use. When statistics are printed, the bus reports the number of transaction and the relative bus utilization in percent.

6 Main Memory Controller

The memory controller model is based on a relatively simple in-order memory controller without write bypassing. It accurately models all address and data bus contention, supports multiple banks of SDRAM or Rambus DRAM chips and handles coherency responses. The following table summarizes the configuration parameters.

Parameter	Description	Default
mmc_simon	enables detailed memory controller simulation	1 (on)
mmc_latency	memory access latency if detailed simulation is off	20 cycles
mmc_frequency	frequency of memory controller in processor cycles	1
mmc_debug	enable debugging output	0 (off)
mmc_collect_stats	gather statistics	1 (on)
mmc_writebacks	maximum number of outstanding writebacks	one from each processor or I/O device

Table 14: Memory Controller Configuration Parameters

The first parameter determines whether a detailed memory controller and DRAM simulation is performed, or whether simple fixed-latency memory accesses are modeled. If the detailed simulation is disabled, each memory controller request completes after a fixed number of memory controller cycles. The memory controller frequency is only relevant if the detailed simulation is disabled, it controls the clock cycle time with respect to a processor cycle. For instance, a fixed-latency simulation with a 25 cycle latency and a memory controller frequency of 3 would complete all requests after 75 processor cycles. Note that the fixed latency refers to the time when a request arrives from the bus until its response is arbitrating for the system bus, it does not include bus or cache handling time.

The debugging output flag is currently unused, but it could control existing debugging routine that can be inserted into the code. If no memory controller statistics is desired it can be turned off. The number of writebacks supported by the memory controller determines when the system bus write flow control will be activated to avoid overflowing the write buffer. Note that writes are handled in order with respect to read requests. Nevertheless, since writes do not result in a response, a separate flow control mechanism is needed. The default number of outstanding writes allows one write per CPU or I/O device. It is recommended that this number not be lowered as deadlock may occur.

6.1 Memory Controller Structure

*files: Memory/mmc_init.c, Memory/mmc_bus.c, Memory/mmc_main.c, Memory/mmc_stat.c, Memory/mmc_debug.c
Memory/mmc.h, Memory/mmc_param.h, Memory/mmc_stat.h*

The memory controller model implements a coherency scoreboard and a queue for pending requests. Requests enter both queues when they are received from the system bus, except writebacks which are non-coherent and are only entered into the pending queue. From the pending queue, requests are forwarded to the DRAM bank controller where they are buffered and processed. The DRAM controllers can return data to the memory controller in critical-word first order, which allows the memory controller to arbitrate for the bus as soon as the DRAMs start returning data.

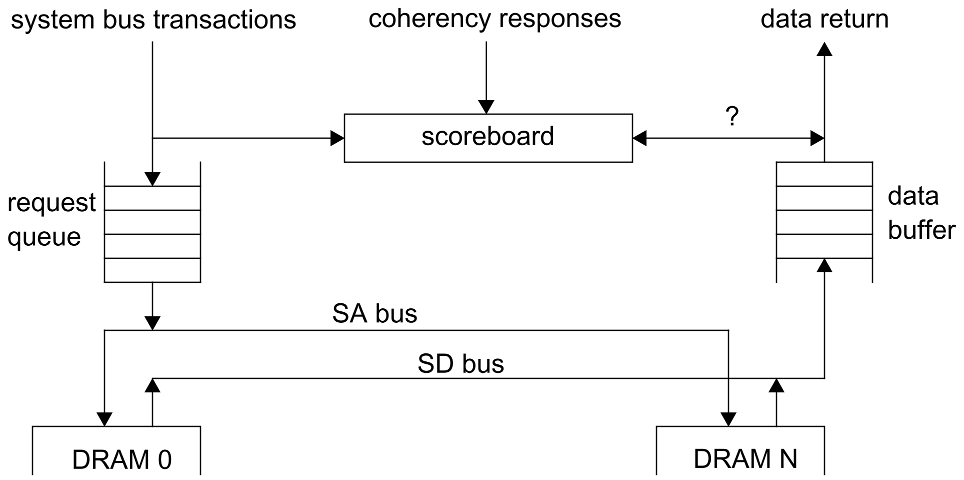

Figure 8: Memory Controller Structure

The number of DRAM banks as well as the interleave factor is configured in the DRAM model, the memory controller communicates with the DRAM banks over a single set of address and data buses of configurable width. The memory controller assumes that the individual DRAM bank controllers provide buffering for requests and data, thus decoupling the DRAMs somewhat from SA/SD bus contention.

6.2 Request Handling

Incoming requests are first processed by the bus interface module in the *MMC_recv_request* routine. This routine initializes the coherency response counter for the transaction, enqueues it in the scoreboard and calls *MMC_recv_trans* to create and enqueue an internal transaction in the request queue. If the queue was empty, the transaction is immediately forwarded to the DRAMs via the SA bus by the *MMC_process_waiter* routine, otherwise the same routine is already scheduled to execute at a later time to remove the next pending request from the queue. The width of the SA bus determines the number of cycles required. It is assumed that the DRAM controllers provide separate buffering for requests.

When a DRAM access starts returning data, the routine *MMC_dram_data_ready* is called to indicate that the memory controller may start arbitrating for the system bus. If all coherency responses have arrived and no cache has a modified copy of the requested data, the memory controller enqueues the transaction in the outgoing buffer and starts arbitrating for the system bus if the queue was empty.

The routine *MMC_dram_done* is called by the DRAM simulator when the data transfer is complete. At this time the internal request data structure is freed and the statistics counters are updated.

Writeback and *writepurge* transactions are handled similarly, except that they do not generate data returns, and that the DRAM timing is different. In addition, *writeback* transactions are not enqueued in the scoreboard, since they are not snooped by the caches.

If detailed memory controller simulation is turned off, the routine *MMC_nosim_start* is called to schedule a completion event after a fixed number of cycles. This routine bypasses the SA and SD bus and DRAM controllers. The routine *MMC_nosim_done* removes the oldest request after a fixed latency and completes it. If additional requests are found in the request queue, it reschedules itself to process the next request after the same fixed latency. In effect, requests are queued as they come in, but each request incurs a fixed and identical latency after it has reached the head of the request queue.

6.3 Coherency

All coherent bus transactions are entered into a scoreboard by the memory controller bus interface, and are removed from the scoreboard when the transaction completes. Initially, the coherency response counter is set to the number of coherent bus modules, and the data return flag and cache-to-cache transfer flag are cleared. As coherency responses come in from bus modules via the *MMC_recv_cohe_response* routine, the coherency counter is decremented. Coherent bus modules may set the cache-to-cache transfer flag at any time. When the DRAM controllers start providing data, the data return flag is set. When the coherency counter reaches zero and the data return flag is set, the memory controller inserts the request in the outgoing queue and arbitrates for the system bus. However, if the request has received a cache-to-cache copy, the memory controller calls *Bus_send_cohe_completion* instead to signal to the requesting module that all snoop responses have been received. In this case the DRAM data response is dropped since another coherent bus module has already provided the data.

6.4 Statistics

The memory controller counts the number of transactions it processes and classifies them into read, write, upgrade and writebacks. For each transaction it records the time it was busy processing it, as well as the latency of read requests. All statistics, including the total number of transactions, is printed to the statistics file at the end of a simulation period.

7 DRAM Model

The simulator supports two different models of DRAM banks: SDRAM and Rambus. Both models include simple controllers that are designed to work with the memory controller described before. In particular, it is assumed that all DRAM banks share one address and data bus to the memory controller, and that the DRAM controllers provide sufficient buffering of requests and data returns to avoid bus conflicts.

The DRAM model requires various configuration parameters, depending on the type of DRAM simulated. The following table summarizes all parameters.

Parameter	Description	Default
dram_simon	enables detailed DRAM simulation	1 (on)
dram_latency	DRAM access latency if detailed simulation is off	18 cycles
dram_frequency	frequency of DRAMs controller in processor cycles	1
dram_scheduler	enable DRAM scheduling, affects open/close row policy	1 (on)
dram_debug	enable debugging output	0 (off)
dram_collect_stats	collect and print statistics	1 (on)
dram_trace_on	print trace information	0 (off)
dram_trace_max	maximum number of trace statements	0 (no maximum)
dram_trace_file	name of trace file	dram_trace
dram_sa_bus_cycles	number of SA bus cycles per address transfer	1
dram_sd_bus_cycles	number of SD bus cycles per data item	1
dram_sd_bus_width	width of SD bus in bits	32
dram_num_smcs	number of DRAM bank controllers	4
dram_num_databufs	number of data buffers/multiplexers	2
dram_critical_word	enables critical-word first transfers	1 (on)
dram_num_banks	number of DRAM banks	16
dram_banks_per_chip	number of internal banks per DRAM chip	2
dram_bank_depth	size of request queue per bank	16
dram_interleaving	interleaving scheme (0: block/modulo, 1: page/modulo, 2: block/sequential, 3: page/sequential)	0
dram_max_bwaiters	maximum number of outstanding DRAM requests	256
dram_hot_row_policy	row open/close policy (0: always close, 1: always open, 2: predict)	0
dram_width	external width of DRAM bank	16
dram_mini_access	minimum size of a DRAM access	16

Table 15: DRAM Configuration Parameters

Parameter	Description	Default
dram_block_size	size of an interleave block	128
dram_type	SDRAM or RDRAM	SDRAM

Table 15: DRAM Configuration Parameters

The first parameter determines whether a detailed timing simulation is performed for each DRAM access or not. If it is turned off, each DRAM access takes a fixed number of cycles as specified by the second parameter. Each cycle is a multiple of processor cycles, given by *dram_frequency*.

If the DRAM scheduler is disabled, part of the DRAM timing model is bypassed, resulting in a fixed latency DRAM access, while still modeling bus contention. The number of SA and SD bus cycles determines how for many cycles a data item occupies a bus. In the case of the SA bus, a request is transferred as one unit. On the SD bus, the number of items is the total size divided by the width of the bus.

The physical configuration of the DRAMs is determined by the number of SMCs (bank controller), data buffers/multiplexers and banks. It is possible that each SMC controls multiple banks, which may or may not share data buffers. In any case, however, all SMCs are connected to the memory controller over one SA bus, and all data buffers share an SA bus to return data to the memory controller. The number of internal banks affects only the modeling of individual DRAM accesses.

The DRAM controller supports four different interleaving schemes. Sequential schemes assign consecutive numbers to adjacent banks. For instance, if an SMC controls two banks, they would be number consecutively under the sequential scheme. Modulo interleaving assigns consecutive banks to banks on different SMC, if possible. Both interleaving schemes can use either block-level or DRAM-page level interleaving. The block size can be configured separately, it is intended to be equal to the L2 cache line size. The DRAM page size depends on the DRAM type.

The hot row policy determines whether DRAM rows (pages) are left open, or are closed after each access. Alternatively, a predictor may be used to dynamically decide to leave a row open.

7.1 DRAM Controller Structure

files: DRAM/dram_init.c, DRAM/dram_main.c, DRAM/dram_refresh.c, DRAM/dram_debug.c, DRAM/dram_stat.c
DRAM/dram.h, DRAM/dram_param.h, DRAM/cqueue.h

The DRAM subsystem consists of bank controllers (SMCs), data buffers/multiplexers and DRAM banks. Each bank controller may control multiple DRAM banks, which in turn may share a data multiplexer. The following figure shows an 8 bank configuration with 2 SMCs and 4 data buffers.

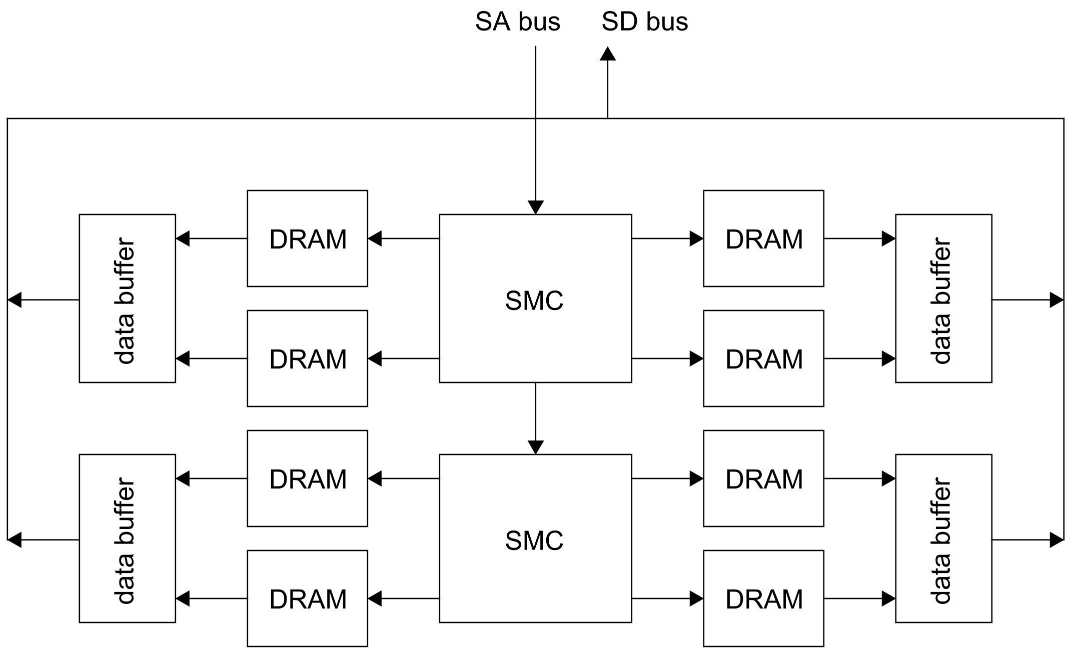

Figure 9: Example DRAM configuration

The DRAM model supports two different DRAM types, plus a simple fixed-latency DRAM model that ignores DRAM chip internal timing details. Both SDRAM and Rambus models model the internal timing of row open/close commands, refresh cycles and contention for external data buses. The following table lists the SDRAM configuration parameters.

Parameter	Description	Default
sdram_tccd	CAS to CAS delay	1
sdram_trrd	bank to bank delay	2
sdram_trp	precharge time	3
sdram_tras	RAS latency, row access time	7
sdram_trcd	RAS to CAS delay	3
sdram_taa	CAS latency	3

Table 16: SDRAM Configuration Parameters

Parameter	Description	Default
sdram_tdal	data-in to precharge time	5
sdram_dpl	data-in to active time	2
sdram_tpacket	number of cycles to transfer one ‘packet’	1
sdram_row_size	size of a DRAM row/page	512
sdram_row_hold_time	maximum time to keep a row open	750000 cycles
sdram_refresh_delay	duration of an auto-refresh cycle	2048 cycles
sdram_refresh_period	auto-refresh period	750000 cycles

Table 16: SDRAM Configuration Parameters

Most parameters are expressed as cycles in terms of *dram_frequency*, which is a multiple of processor cycles. Ideally, the DRAM frequency is set to a native SDRAM frequency (e.g. 100 or 133 MHz), and all timings are expressed in DRAM cycles.

The following table summarizes the configuration parameters for RDRAMs.

Parameter	Description	Default
rdram_trc	delay between ACT commands	28
rdram_trr	delay between RD commands	8
rdram_trp	delay between PRER and ACT command	8
rdram_tcbub1	read to write command delay	4
rdram_tcbub2	write to read command delay	8
rdram_trcd	RAS to CAS delay	7
rdram_tcac	CAS delay (ACT to data-out)	8
rdram_tcwd	CAS to write delay	7
rdram_tpacket	number of cycles to transfer one ‘packet’	4
rdram_row_size	size of a DRAM row/page	512
rdram_row_hold_time	maximum time to keep a row open	750000 cycles
rdram_refresh_delay	duration of an auto-refresh cycle	2048 cycles
rdram_refresh_period	auto-refresh period	750000 cycles

Table 17: RDRAM Configuration Parameters

7.2 DRAM Operation

The DRAM model receives requests from the memory controller by calling the routine *DRAM_recv_request*. This routine models SA bus contention and inserts requests in a queue if the SA bus is currently busy, otherwise it sends the request directly to the appropriate SMC. If the SA bus is busy, an event handler will remove the next oldest request from the queue at a later time and send it to one of the SMCs.

At the SMC (slave memory controller), requests are either queued if the target bank is busy, or a bank access is initiated. The *DRAM_access_bank* routine chooses either one of the two detailed DRAM models, or the simple fixed-latency DRAM model to handle the request. The DRAM specific routines calculate the first cycle that data is available based on the current open row, precharge state, and possible refresh cycles, calculate and record the occupancy of the DRAM, and update other internal state. As a result, an event is scheduled that forwards data to the data buffers and initiates the next DRAM access, if any are pending. If the data buffer is idle, data is returned to the memory controller over the SD bus, otherwise the transaction is enqueued in the data buffers.

Both SDRAM and RDRAM model the auto-refresh capability of modern DRAMs. The initial refresh cycle is staggered across all banks, and is repeated periodically for all rows. If a refresh is required, it is either initiated immediately if the bank is idle, or a flag is set and the refresh is started after the current transaction completes. Each refresh cycle takes a configurable number of cycles.

Part IV: I/O Subsystem

1 I/O Devices

This section describes the generic bus interface for I/O devices. The interface does not implement any I/O device functionality, as this is expected to be provided by callback functions and auxiliary data structures. The bus interface supports uncached reads and writes that target the I/O device as well as all coherent transactions issued by CPUs. For uncached reads, the bus interface can automatically issue the reply-transaction if the attached I/O device does not do this on its own. The I/O device can issue uncached read and write transactions to communicate with other I/O devices and the CPUs system control module. For coherent DMA operation, the bus interface supports *read_current*, *read_sh* and *read_own* transactions. *read_own* implies that the device intends to modify part of the cache line, the bus interface can automatically issue a *writeback* transaction upon receiving the data reply. Finally, *writepurge* transactions can be used to write entire cache lines and purge the line from all third-party caches.

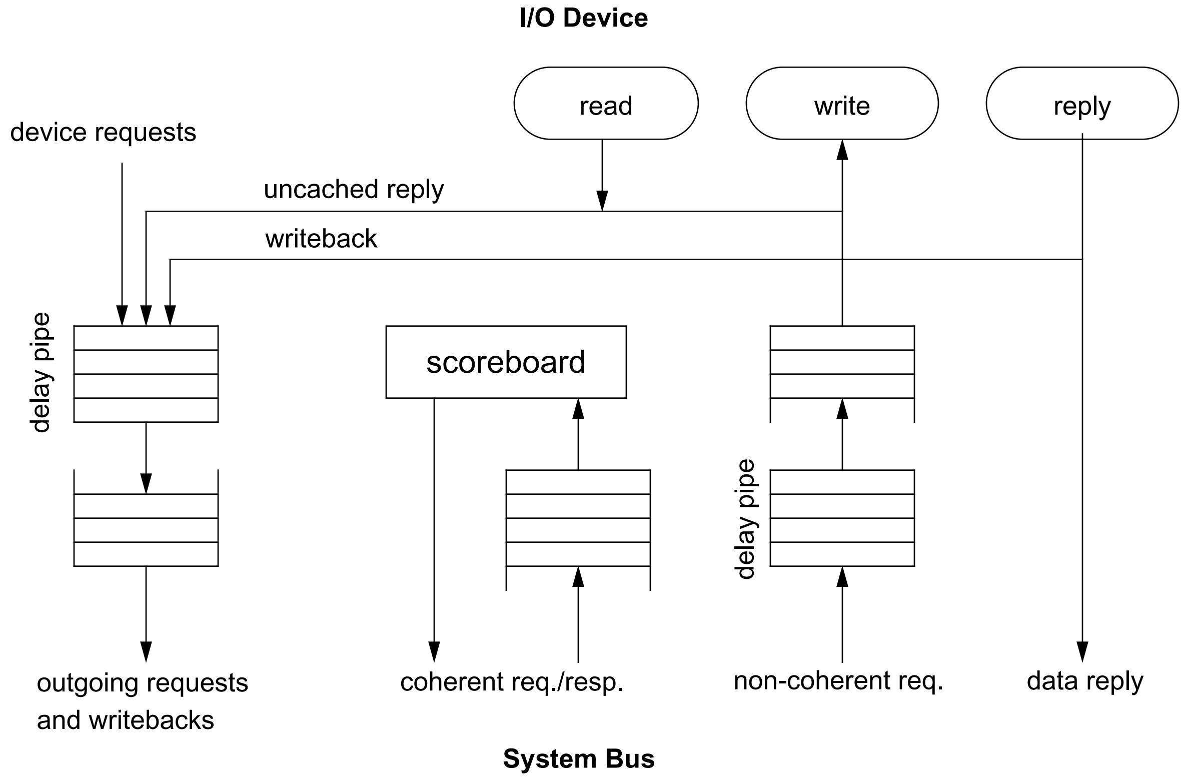

Figure 10: I/O Device Bus Interface

Incoming requests are buffered in two separate queues. Coherent requests are always broadcast to all processors and coherent I/O devices by the bus, these requests are buffered in the coherent queue. Non-coherent requests (*read_uncached*, *write_uncached*) are targeted at a particular device and go through a delay pipeline before being buffered in the input queue. The delay pipeline models the delay incurred in the I/O bridge and I/O bus, the pipeline depth can be configured. Outgoing requests, data responses (for uncached reads) and writebacks (including upgraded cache-to-cache transfers) are buffered in the outqueue. These requests are also passed through a delay pipeline to model the delay incurred in an I/O bridge.

The scoreboard is an array of outstanding requests and writebacks, and may include requests that are buffered in the upper part of the I/O device. Requests are inserted into the scoreboard when they reach the top of the coherent queue, and are removed when they receive the corresponding data return or when the modified data is written back through a *writeback* transaction.

1.1 Initialization

files: IO/io_generic.c, Caches/system.c
IO/io_generic.h

Any I/O device should provide an initialization function that creates and initializes data structures for the I/O device in each node. This function should register the devices physical address space with each nodes address map (via *AddrMap_insert*), allocate physical memory for these addresses and install the appropriate simulator page table mappings (via *PageTable_insert* or *PageTable_insert_alloc*). It then calls the bus interface initialization function *IOGeneric_init* with callback functions for read and write accesses and data replies as arguments. The generic initialization function sets up an array of I/O device pointers (one per node) and initializes the various queues.

Parameter	Description	Default
io_latency	I/O bridge latency (depth of delay pipelines)	1
bus_io_requests	number of outstanding I/O device requests	4
bus_total-request	number of coherent requests per bus	8

Table 18: I/O Device Configuration Parameters

The *io_latency* parameters controls the depth of the delay pipelines, which model the additional I/O transaction latency incurred by an I/O bridge. The number of I/O requests determines the depth of the non-coherent queue. The total number of outstanding requests determines the depth of the coherent queue, both in the processors and the I/O devices.

1.2 Handling Incoming Bus Transactions

files: IO/io_generic.c
IO/io_generic.h

The bus module calls *IO_get_request* or *IO_get_cohe_request* when it is transferring a bus transaction to an I/O device. These functions enqueue coherent transactions in the coherent request queue and places non-coherent requests in the delay pipeline. If any of the input queues or the delay pipeline where empty before the current request was handled, it schedules an event for the next bus cycle to handle any requests. The event handler will automatically reschedule itself after completion if more requests are waiting.

The input event handler first handles coherent requests. All requests that were issued by this device are inserted into the scoreboard. The routine then performs a scoreboard lookup. If no matching request was found, the request was issued by this device or it matches a pending *read_current*, an *excl* snoop response is issued, indicating that the request can be satisfied in exclusive mode from this modules perspective. If a snooped *read_sh* matches a pending *read_sh*, the snoop response is *shared*. If a snooped read request matches a pending writeback, the writeback transaction is upgraded to a cache-to-cache transfer and an *excl* snoop response is issued. For all other combinations of snooped request and scoreboard match the coherent queue is stalled.

After processing the coherent queue, the routine handles requests from the non-coherent queue. First, the routine performs the memory operations associated with the request (and possibly any coalesced requests). The routine then calls the read or write callback function if it exists. The return value of this function indicates if the request has been handled by the upper part of the I/O device. If the callback function returns 1, or no function is registered, the bus interface removes the request from the input queue, frees up the request structure or issues the uncached reply transaction.

After handling at most one coherent request and one non-coherent request, the routine checks if more requests are waiting in either incoming queue or delay pipeline and reschedules the event if this is the case.

1.3 Outgoing Bus Transaction Handling

files: IO/io_generic.c
IO/io_generic.h

The I/O device can call *IO_start_transaction* with the request as argument to initiate a bus transaction. The following fields in the request structure must be set correctly:

- src_node, src_proc (module ID)
- dest_node, dest_proc (destination module - based on address map lookup)
- paddr - physical address
- size - request size in bytes
- type - request/reply
- req_type - write_uc, reply_uc
- prcr_req_type - read/write

This routine inserts the request into the device queue and schedules an event for the next bus cycle to handle, unless the queue or pipeline had already requests waiting.

The event handler first checks for requests in the out-queue. If a request was waiting, a bus transaction is initiated by calling *IO_start_send_to_bus* and the request is removed from the queue.

If the system interface is already owning the bus and the bus is idle, it sends out the request immediately, otherwise it places the request in a request queue and arbitrates for the bus. The function *IO_in_master_state* is called at the beginning of a bus transaction, either by the bus module or the system interface. It computes the number of bus cycles for this transaction and marks the bus busy for this time. At the end of the transaction, the bus module calls *IO_send_on_bus*, which sends the request to the target module. If the current transaction is a writeback, it is removed from the scoreboard.

After handling a request from the output queue, the event handler moves the oldest request from the delay pipeline to the output queue, moves the oldest entry from the device queue into the delay pipeline and reschedules itself if either queues or the pipeline are not empty.

1.4 Data Reply Handling

files: IO/io_generic.c
IO/io_generic.h

Incoming data replies are handled by the routine *IO_get_reply*, which is called by the bus module at the end of a data transaction. This routine first performs the memory operation associated with the request and calls the reply callback function if it exists. If the request was a *read_current* or *read_shared* and the callback function returned 1, it is removed from the scoreboard and the request structure is returned to the pool. A *read_own* request is issued when the I/O device intends to modify a portion of a cache line. When the corresponding reply is received and the callback function returned 1, the bus interface issues a *writeback* transaction. Note that the request remains in the scoreboard and may be upgraded to a cache-to-cache transfer if another module issues a conflicting read.

1.5 Request Completion for I/O Devices

files: IO/io_generic.c, Processor/pagetable.cc
IO/io_generic.h

Each request contains pointers to two functions that will be called upon request completion. In case of requests issues by I/O devices, only one of these function is currently used. The generic I/O device layer provides routines that move different amounts of data between main memory and I/O buffer space. The read routines copy a byte, word or a block of data from main memory into the buffer (the buffer pointer is part of the request structure). Write routines behave the opposite way, they write data from the buffer to main memory. The transfer size for block transfer routines is specified in the request size field.

2 Realtime Clock

The realtime clock chip is modeled after several existing chips, like the Dallas DS1743 or SGS M48T02, but it is not an exact model of any particular chip. In particular, it does not model the battery-backed SRAM part of these chips.

2.1 Operation

The clock chip implements a realtime clock with a one second resolution, and contains two programmable periodic interrupt sources with a minimum interval of 1 ms. The following table summarizes the register set. All registers are 8 bit wide.

Address	Description	Bits<7:4>	Bits<3:0>
0xFFFF F000	Status Register	n/a	n/a<3:2> int2 int1
0xFFFF F001	Control Register	n/a	n/a<3:2> write read
0xFFFF F002	Interrupt 1 Control	n/a<7:5>	10s 1s 100ms 10ms 1ms
0xFFFF F003	Interrupt 1 Vector	vector<7:4>	vector<3:0>
0xFFFF F004	Interrupt 1 Target	target<7:4>	target<3:0>
0xFFFF F005	Interrupt 2 Control	n/a<7:5>	10s 1s 100ms 10ms 1ms
0xFFFF F006	Interrupt 2 Vector	vector<7:4>	vector<3:0>
0xFFFF F007	Interrupt 2 Target	target<7:4>	target<3:0>
0xFFFF F008	Year (00-99)	10 years	year
0xFFFF F009	Month (01-12)	10 months	month
0xFFFF F00A	Date (01-31)	10 dates	date
0xFFFF F00B	Day of Week (01-07)	00	day of week
0xFFFF F00C	Hour (00-23)	10 hours	hour
0xFFFF F00D	Minutes (00-59)	10 minutes	minute
0xFFFF F00E	Seconds (00-59)	10 seconds	seconds

Table 19: Realtime Clock Registers

The status register indicates if and which interrupts are pending. Writing a 1-bit into bit 1 or 0 clears the interrupt. If either the read and write bit in the control register is one, the clock registers are not updated. The read bit should be set before the clock registers are read, to avoid reading an inconsistent time, and should be cleared after reading the clock registers. The write bit inhibits updating of the clock registers, similar to the read bit. In addition, it allows software to write the clock registers in order to set the clock. When the write-bit is cleared, the contents of the clock registers is written to the realtime clock.

The interrupt control registers specify the interval for the periodic interrupt. In each register, more than one bit may be set. At the specified interval, the chip will issue an interrupt transaction to the

CPU specified in the target register, using the value in the vector register as interrupt vector. If the target ID is 0xFF, the interrupt will be sent to all CPUs in the node. Note that no interrupt will be triggered if the corresponding interrupt bit in the status registers is not cleared.

The clock registers specify the time in BCD format. They are updated every second, unless the read or write bit is set. When the realtime clock chip is initialized, the internal clock is set to the host systems wall time. It is then incremented according to the simulation time and the systems clock frequency.

2.2 Initialization

files: *IO/realtime_clock.c*
IO/realtime_clock.h

The system initialization function *SystemInit* calls the clock chips initialization function *RTC_init*. This function creates an array of clock data structures, one for every node. It then inserts a page of physical memory that contains the clock registers in the nodes page table and registers the address range in each nodes address map. The internal clock counter is initialized with the host systems current clock, and the clock registers are set to the current time. Finally, the initialization function creates a simulator event that is scheduled every millisecond relative to simulator time.

2.3 Clock Event

files: *IO/realtime_clock.c*
IO/realtime_clock.h, Caches/syscontrol.h

When the clock event occurs the simulator calls the *RTC_update* function. This function increments the internal clock counter by one millisecond and checks if an interrupt needs to be triggered. If this is the case, and the corresponding interrupt bit is cleared, it issues an uncached write transaction to the target processors interrupt register in its system control space and sets the clock interrupt bit. After 1000 ms, the clock registers are updated, unless the read or write bit is set.

2.4 Reading and Writing Registers

files: *IO/realtime_clock.c*
IO/realtime_clock.h

The I/O device bus interface calls the realtime clocks *RTC_read* function if it received a read transaction to the clock chips address space. The bus interface has already performed the read operation, the call-back function only changes the request into a reply and sends it to the I/O bus interface via *IO_start_send_to_bus*.

Upon receiving a write transaction, the bus interface calls *RTC_write*. Since writes to the physical page are performed in the I/O bus interface, this function can ignore most of the transactions. Only if the transaction modifies the control register, it compares the new value to the previous value. If the bit has been cleared, it updates the clock counter with the values in the clock registers.

2.5 Statistics

files: *IO/realtime_clock.c*
IO/realtime_clock.h

The realtime clock measures interrupt latency for all interrupts it generates and records the minimum, maximum and average latency it observes. Latency is defined as the time from the moment when the clock sends the interrupt transaction to the bus interface until the CPU resets the interrupt bit in the control register.

Upon completion of a simulation run, the realtime clock model prints its configuration which includes the interrupt period settings, interrupt target and interrupt vector for both possible interrupt sources. It then prints the interrupt count and interrupt latencies in milliseconds or microseconds.

3 PCI Subsystem

The PCI subsystem consists of a PCI bridge which manages the PCI configuration spaces for all PCI devices. The simulator currently does not model a PCI bus, hence the devices themselves are attached directly to the system bus (via the generic I/O bus interface).

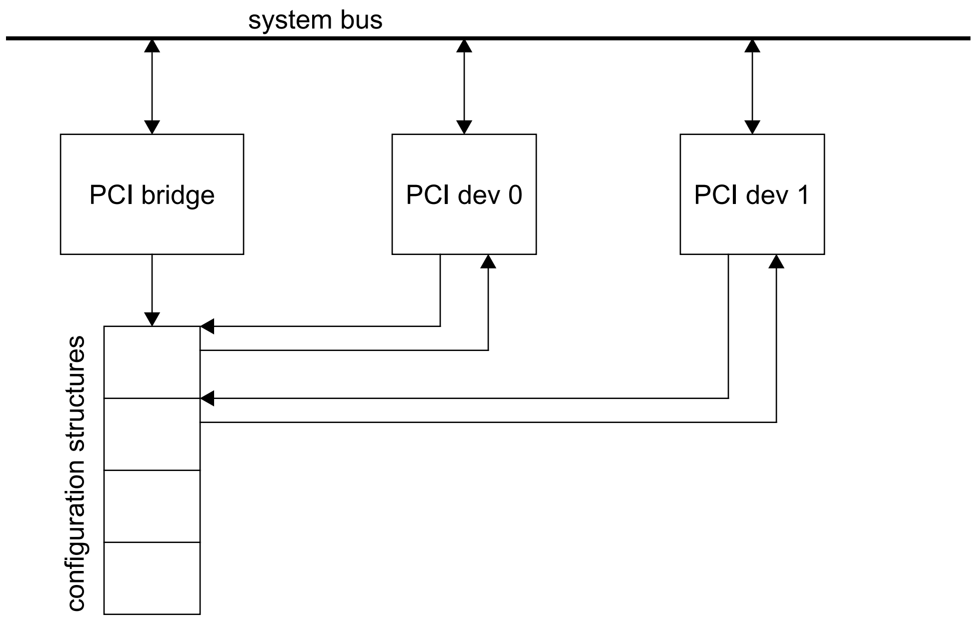

Figure 11: PCI Bridge and PCI Devices

The PCI bridge maintains an array of PCI device configuration spaces which can be accessed directly from the system bus. Each PCI device registers with the PCI bridge by calling the routine *PCI_attach*. This routine assigns a PCI slot to the device (and hence a configuration space) and stores the bus module number of the attached device in a separate array. In addition, each device provides a callback function that returns the required address spaces. The attach-routine returns a pointer to the first PCI configuration structure for this device. The device is expected to fill in the configuration space for every supported function with the correct information.

3.1 PCI Bridge Initialization

files: IO/pci.c
IO/pci.h

Initialization of the PCI bridge is split into two distinct routines. Before any devices can be attached to the PCI bridge, the configuration space array must be allocated and initialized (routine *PCI_init*). After all PCI devices have been initialized (and attached to the bridge), the bridge itself can be attached to the system bus (routine *PCI_BRIDGE_init*). This is necessary because the PCI bridge is a non-coherent device, and its bus module number must be higher than the module number of any coherent (snooping) device. Since bus module numbers are assigned in order, the bridge must be attached after at least all coherent devices.

Attaching the PCI bridge to the bus is done by first calling the generic I/O bus interface initialization routine and then inserting the configuration space into the address map and simulator page table.

3.2 PCI Device Registration

files: IO/pci.c
IO/pci.h

PCI devices attach to the bridge by calling the routine *PCI_attach*. This routine takes as arguments the node number and bus module number of the device and a callback function that is used to configure the PCI address spaces. The routine then assigns a PCI slot (and hence a configuration space) to the device, stores the bus module ID and callback function is separate data structures and returns a pointer to the first configuration structure for this device. The PCI device should then fill in the configuration spaces for all supported functions.

3.3 PCI Bridge Operation

files: IO/pci.c
IO/pci.h

The PCI bridge, just like other I/O devices, uses the generic I/O bus interface to communicate with other bus modules. The read callback function merely initiates a response transactions since the actual access to the physical address location is performed by the I/O bus interface.

The write callback function needs to perform some special processing on writes to any address space configuration registers. If the value written to such a register is -1, the bridge calls the corresponding devices address mapping callback function, passing the register value, node and module ID, function and register number as arguments.

The callback function is expected to examine the function and register number and store the minimum address space size in a pointer that is provided. In addition, it returns the appropriate PCI address space flags (I/O or memory space, memory space attributes) in a second pointer.

The PCI bridge converts the returned size into a bit mask as specified in the PCI documentation and writes the mask into the configuration register. The PCI configuration software will then read the required address space sizes from all devices, determine a feasible address assignment and write the base addresses in the configuration registers. At this point, the PCI bridge calls the devices callback function again, but with a register value other than -1. This indicates to the device that the address space is to be mapped at the specified base address. The PCI bridge then merges the address space flags into the lower bits of the base address.

The routine *PCI_print_config* takes as argument a pointer to a PCI configuration space (the return value of *PCI_attach*) and prints configuration information for all valid functions in the device configuration space. This includes device and vendor ID, device class, interrupt settings and address space assignments. This routine should be called from the associated device during the statistics output phase.

4 SCSI Subsystem

The SCSI subsystem consists of SCSI bus model and attached SCSI devices, of which one is the designated host controller. The following figure shows the phases of a SCSI bus transaction as it is modeled, as well as the relevant timing parameters.

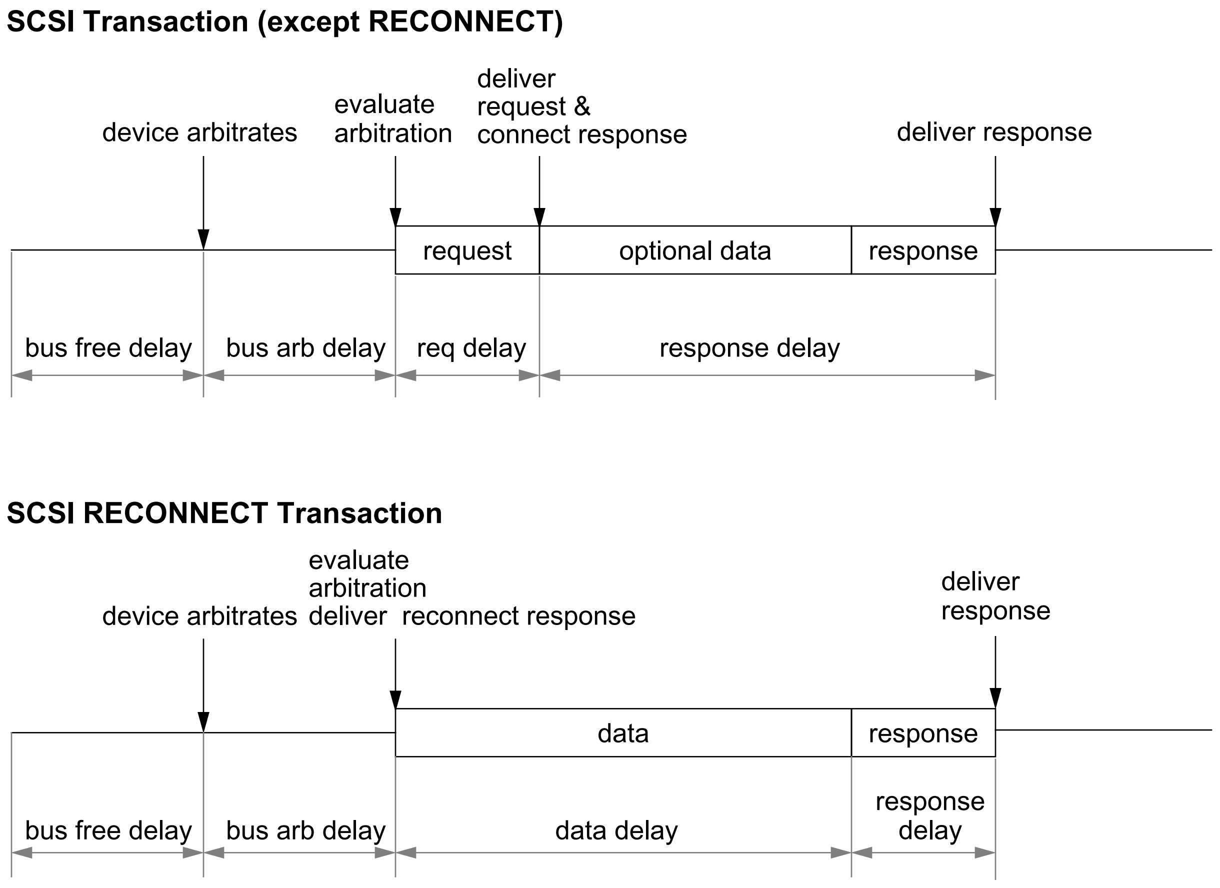

Figure 12: Simplified SCSI Protocol

The *bus free delay* is the minimum amount of time between the end of a transaction and the start of the next arbitration phase. The *arbitration delay* is the amount of time that devices must assert their arbitration requests before the next bus owner is determined, it is essentially the delay between an arbitration request and the start of the next transaction. In the simulation model, devices may set arbitration requests at any time, but the bus will not consider the request until either *bus-free* + *bus-arb-delay* after the next transaction, or *bus-arb-delay* cycles after the first arbitration request on an idle bus.

The *request delay* determines how long after the start of a transaction the request is delivered to the target device. It is the sum of the delay between the end of the arbitration phase and the beginning of the request block transfer and the request transfer time. In the simulation model this delay is approximated by a fixed time.

The *response delay* determines the time between receiving a request and sending a response. It is controlled by the target device, and in case of a read transaction that is satisfied with a data transfer includes the time needed to transfer the data. The following table summarizes the SCSI bus parameters that effect all devices.

Parameter	Description	Default
scsi_width	width of data path in bytes	2
scsi_frequency	bus frequency in MHz	10
scsi_arb_delay	arbitration delay in SCSI bus cycles	24 (2.4 µs)
scsi_bus_free	bus free delay in SCSI bus cycles	8 (800 ns)
scsi_req_delay	arbitration to request delivery delay in cycles	13 (1.3 µs)
scsi_timeout	device selection timeout	10000 (1 ms)

Table 20: SCSI Bus Parameters

Unlike the system bus frequency, the SCSI bus frequency is specified in MHz. The simulator uses the *clk-period* parameter to convert this into a cycle-ratio value. All delay parameters are specified in SCSI bus cycles. The timeout default value is smaller than the recommended value (250 ms) to reduce simulation time for the rare case that a SCSI transaction times out. The SCSI bus supports as many devices as the data path has bits, for instance a 2 byte wide bus thus supports up to 16 devices. The device with the highest ID has the highest arbitration priority, this is usually the host adapter.

4.1 Data Structures

The SCSI request data structure contains the initiator and target IDs, the request/response type and several argument fields, such as logical unit number, block number and transfer size. These structures are managed in a pool, similarly to the system bus requests.

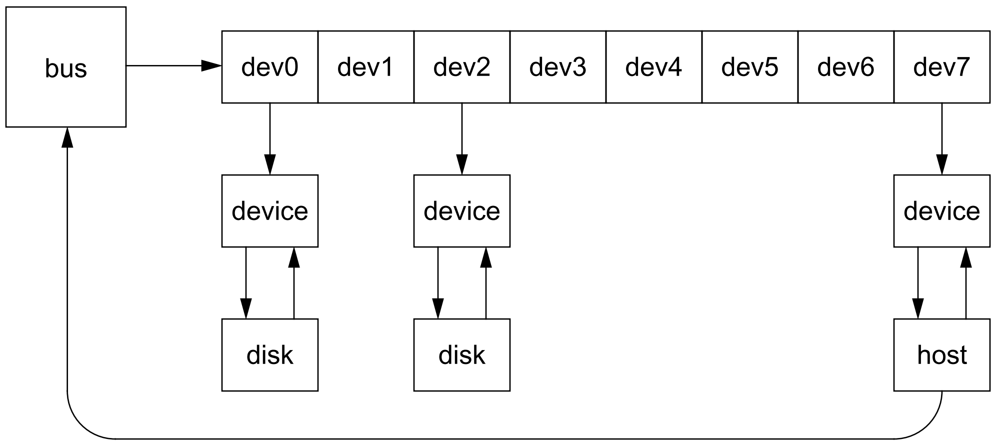

Figure 13: SCSI Bus Control Structures

The SCSI bus structure contains a pointer to the current request, a state indicator and a list of attached devices. Each device is described by a structure that contains fields for the arbitrating request, a pointer to the device-specific control structure and pointers to callback functions for request/response delivery and statistics that are provided by the particular device.

4.2 SCSI Requests

files: IO/scsi_bus.h

SCSI request structures are managed in a pool by the simulation library, similar to the system bus request structures. The request structures contains elements that describe the SCSI request to the target device. The request type identifies which operation the target is to perform when receiving the request. Currently, the following requests are supported:

- miscellaneous - various commands that don't transfer data and usually don't require special processing by the device, such as 'disable media change'
- inquiry - read device identification
- mode_sense - read device identification
- read_capacity - read the device capacity
- request_sense - read sense data from device after an error occurred
- read, write - read or write N blocks of data
- prefetch - read data into the device cache but don't return data
- seek - position head
- format - format media
- sync_cache - synchronize on-board cache by writing back all dirty data
- reconnect - target reconnects when a long-latency operation is complete

The *reply-type* field is used by the target when responding to a request. Note that a request might receive several responses. An initial response is returned after the bus model has called the target devices request callback function, which is expected to set the reply-type and the bus-cycles fields appropriately. This response indicates that the target has received the request. If the target does not disconnect, the connect-response signals the beginning of the data transfer phase to the initiator. Other possible responses are *busy*, *reject* or *disconnect*.

After returning this response to the initiator, the bus model schedules the final response after N bus cycles as specified in the request structure. This response normally signals the end of the data transfer phase, it may be a *save_data_ptr* or *complete* response.

The initiator can return the request structure to the pool when it receives the second response, the target device must retain a copy of the structure if necessary. Additional responses can be send by the target when it needs to transfer more data, or when the operation is complete. The response structures are also returned to the pool by the initiator (which is the sink of the response transaction). The delay incurred by response transactions is always specified in the bus-cycles field. The following responses are supported:

- complete - operation has completed
- connect - target has accepted request and starts transferring data
- disconnect - target accepts the request but releases the bus
- save_data_pointer - abort current data transfer, more data will be transferred later
- busy - device can not accept request temporarily
- reject - request is illegal for this device
- timeout - device does not respond

Each request is identified by a *initiator-target* pair of device IDs. The *initiator* issues the original request to the target, the *target* sends one or more response transactions to the *initiator*. Additional parameters for most requests include the *logical unit number* (LUN), the *logical block address* (LBA), the request length in blocks, an *immediate*-flag, a queue message and a queue tag. If the immediate-flag is set, the target can signal completion of certain requests (write, seek, prefetch, format, sync-cache) before the operation has been performed. The queue tag indicates if and how the request can be queued. *No-queue* prohibits queueing, if the device is busy or the queue is not empty it returns a busy-status. *Simple-queue* request are enqueued at the end of the device queue. *Order-queue* requests are similar except that the device may not reorder requests in such a way that another request bypasses the ordered request (in either direction). The *head-of-queue* message indicates that the request is to be inserted at the head of the device queue. The queue tag identifies to which queued request a response belongs.

The remaining fields in the request structure do not correspond directly to fields in real SCSI requests. The *orig-request* field is used to store the original request type if the device changes the request type during processing, for instance a read may be converted into a prefetch after completion.

Each SCSI request carries a pointer to a buffer that is used to transfer the actual data. This buffer is also used to hold the information returned by the *inquiry* and *read_capacity* requests. Separate data structures describe the buffer contents for these requests. The *start-block* and *current-length* fields are used by the device while processing the requests.

The *transferred* field is used to keep track of how much data has already been transferred if the transfer happens in more than one transaction. For instance, if the write buffer fills up while the device receives a write request, it returns the *save-data-pointer* status and sets the *transferred* field to the correct value. This allows the initiator for instance to free the corresponding buffer segments. In addition, both parties can keep track of how much more data needs to be transferred.

4.3 SCSI Bus Model

files: *IO/scsi_bus.c*
IO/scsi_bus.h

The routine *SCSI_init* reads the global SCSI parameters described above and stores them in global variables. It should be called before any other SCSI components are initialized. A SCSI bus is created and initialized by the routine *SCSI_bus_init*. It takes the host-node ID and a bus-ID as parameters and returns a pointer to the bus control structure.

The SCSI bus model operates in the three basic phases outlined above. If the bus is idle and a device arbitrates, an event is scheduled to evaluate the arbitration requests after *arb_delay*. The bus module then determines the bus winner and schedules the request delivery event to occur after *req_delay* cycles. The request delivery event handler checks if the target device exists. If not, it converts the request into a timeout response and schedules the response event to occur after *timeout* cycles. If the target device exists, it calls the request callback routine specified for this device. This routine is expected to convert the request into the appropriate response and to set the *buscycles* field in the request structure to the appropriate response delay. The event handler routine schedules the response event after this number of cycles. The response event handler calls the initiators response callback function and then schedules the arbitration event after *BUS_FREE* + *ARB_DELAY* cycles.

The routine *SCSI_bus_stat_report* prints out statistics for the SCSI bus and then calls the statistics callback routines for all attached devices.

4.4 SCSI Devices

In general, most SCSI devices will provide a specialized initialization routine. After it has initialized the device-specific control structure, this routine should call *SCSI_device_init* to initialize the SCSI device bus interface and to attach the device to a specific bus. This routine takes as arguments a pointer to a bus structure, the SCSI device ID, a pointer to the device-specific storage and pointers to four callback functions for request/response and statistics print/reset. The routine returns a pointer to the device control structure.

SCSI requests are issued by calling *SCSI_device_request* with pointers to a SCSI device structure and the SCSI request as arguments. If there is currently no request pending, it places the new request in the devices arbitration pointer. If the SCSI bus is idle, the routine changes the bus state to *arbitrate* and schedules the arbitration event in *ARB_DELAY* cycles.

5 SCSI Host Adapter

The SCSI host adapter is split into two pieces, a bus interface shell and the actual controller module. The bus interface implements the host bus interface (utilizing the generic I/O module) as well as the SCSI bus interface, while the controller implements the device-specific registers and state machines. This organization allows one to implement different controllers with the same basic bus interface framework.

5.1 Generic SCSI Adapter Module

files: IO/scsi_controller.c
IO/scsi_controller.h

The generic SCSI controller implements basic bus interface functionality for both the system bus and the SCSI bus. For each SCSI adapter within a node, the initialization routine first allocates and initializes a generic I/O device module for its system bus interface, and then allocates a controller structure for every controller on the node and initializes it. It also attaches to the PCI bridge and initializes the PCI configuration space. Next, the routine creates a SCSI bus module for each controller, determines its own SCSI ID and attaches itself as a device to the bus. After this, it attaches all other SCSI devices to the bus. Currently, only one or more SCSI disks (specified in the configuration file) are supported as SCSI devices. Finally, the initialization routine calls the controller-specific initialization routines for each controller.

Parameter	Description	Default
numscsi	Number of SCSI adapters per node	1
numdisk	Number of SCSI disks per adapter	1

Table 21: SCSI Controller Parameters

The generic SCSI controller interfaces to the SCSI adapter module through a set of callback routines that are described in a structure. The adapter initialization routine sets up the structure with pointers to the appropriate routines and sets a pointer in the generic SCSI controller to point to this structure. Another pointer points to the control structure used by the SCSI adapter.

The SCSI controller implements a more advanced bus interface that uses the generic I/O module. The bus interface consists of independent queues for replies, DMA requests and interrupts, and a prioritization routine that picks a request from the queues and forwards it to the generic I/O module. The reply queue provides sufficient space to hold a reply for every possible pending read request. The DMA queue and interrupt queue have a fixed size, the SCSI adapter must check the return value of the *SCSI_cntl_host_issue* or *SCSI_cntl_host_interrupt* routines to determine if the request was accepted by the corresponding queue. Whenever a request is inserted into a queue, the multiplexing event is scheduled for the next bus cycle to pick the highest priority request from the queues and forward it to the generic bus interface.

Since the generic SCSI controller module utilizes the generic I/O module as its system bus interface, it must provide callback routines for read and write requests. The read callback routine traverses the list of coalesced requests, calling the controller-specific read-routine for each individual request. Note that the arguments to the controller-specific read-routine are only a pointer to the controller structure, the request address and the request size. The data transfer occurs independently when the request is performed by the generic I/O module. When all coalesced requests have been forwarded to the SCSI controller, the routine changes the request into a reply and inserts the reply into the bus interface reply queue.

The write callback routine also traverses the list of coalesced requests, calling the controller-provided write-routine with the offset and size as arguments. Again, the actual write has already been performed by the generic I/O module, the controller-specific routine can obtain the new value by reading the corresponding registers, or by using *IO_read_** routines.

The routine *SCSI_cntl_host_issue* can be called by the specific adapter to issue a transaction on the system bus. The caller needs to allocate a request structure, fill in the request address and size, processor request type (*read* or *write*), fill in any device-specific fields as well as the completion callbacks. The generic routine then fills in the source and destination module number, determines the correct request type (*read_current*, *read_exclusive* or *write_purge*) depending on request size and address alignment and queues the request in the bus interface DMA queue.

The routine *SCSI_cntl_interrupt* can be used to issue an interrupt transaction on the bus. It gets all necessary parameters such as target processor and interrupt vector from the PCI configuration space, allocates and sets up a system bus request structure and issues it to the bus interface.

Furthermore, as a PCI device, the generic SCSI controller module needs to implement a callback routine for address space mapping. Since address mapping is controller specific, this routine calls the corresponding callback routine for the specific SCSI adapter.

The other responsibility of the generic SCSI controller module is to interface to the SCSI bus. For this purpose it implements several callback routines. The routine *SCSI_cntl_io_wonbus* is called when the controller won arbitration, it is currently empty. The routine *SCSI_cntl_io_request* is called when a device sends a request to the controller. This is currently not supported, the routine writes an error message and exits. The routine *SCSI_cntl_io_response* is called when a SCSI bus response is sent to the controller. It simply calls the controller-provided callback routine which handles the response.

Finally, the generic SCSI controller provides three statistics-related routines. *SCSI_cntl_print_params* is called before the statistics are reported. It first calls the PCI device configuration print routine, next it calls the specific SCSI adapter print routine and finally it calls the SCSI bus print routine. The routine *SCSI_cntl_stat_report* is used to report statistics for the SCSI controller, the SCSI bus and all attached devices. *SCSI_cntl_stat_clear* resets the statistics counters for the SCSI controller, the SCSI bus and all attached devices by calling the respective clear-routines.

5.2 Adaptec AIC7770 SCSI Controller Model

This module models a hypothetical version of a AIC7770-based PCI SCSI adapter. The adapter is hypothetical only in its combination of supported features, the control registers and internal operation are modeled accurately enough to use it with an almost unmodified BSD device driver. The AIC7770 and its descendents are a family of microprogrammed single or dual channel SCSI controllers that support a variety of SCSI standards, including wide and ultra-SCSI. For more details on the AIC 7770 controller chip please refer to the Adaptec databook.

5.2.1 Control Registers

The control registers can be roughly split into the following groups:

- PCI configuration space
- SCSI control
- host interface control
- scratch RAM
- SCSI command blocks

The device implements the full PCI configuration space. It uses only one of the eight possible functions and requires only one I/O address space of 192 bytes. It identifies itself as a SCSI storage class device and requires one interrupt line. The following figure shows the logical organization of the control registers. Note that the PCI configuration registers are logically in a separate address space, since they are not mapped as part of the PCI configuration process, hence they are not shown here.

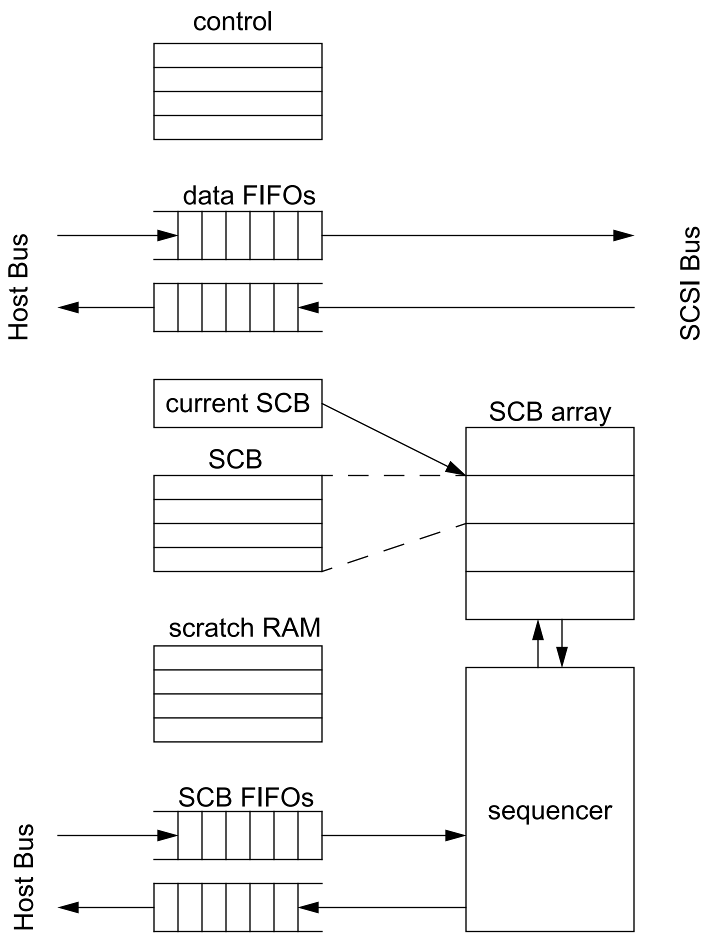

Figure 14: Adaptec SCSI Controller Block Diagram

The SCSI control registers are used to configure the SCSI interface. It allows the device driver to combine two independent channels into one channel of twice the width, restrict the SCSI transfer rate, set the degree of pipelining for synchronous transfers and enable or disable certain SCSI interrupt conditions. Most of these registers have no effect on the simulation model, since the SCSI bus configuration is defined by parameters from the configuration file. In other words, the SCSI adapter will perform data transfers based on the SCSI bus width and frequency that is specified in the configuration file, regardless of the settings in the control registers. Other registers that allow software to directly control the SCSI bus are not supported by the adapter model, since these registers are not used for normal operation. Interrupt settings, however, are honored. The SCSI adapter generates an interrupt only if the corresponding bit is set in the interrupt enable register.

The host interface control registers determine the behavior of the host bus interface, such as data buffer full and empty thresholds for DMA transfers and interrupt settings. Again, only a subset of these registers is supported. Since the simulation model does not model data transfers at a byte level, the data FIFO and the corresponding control registers are not modeled at all.

The adapter provides space for a number of SCSI command blocks (SCBs), each corresponding to an outstanding command. At any given time, only one SCB is mapped into the control register set. The *scb-pointer* register is used as an index into the SCB array. Whenever this pointer is changed, a different SCB is mapped into the SCB registers. This, however, does not affect operation of the sequencer. Another register *scb-cnt* specifies the byte offset into the current SCB and can be used to read or write fields in the SCB. Commands are issued to the SCSI adapter by first setting up an SCB entry and then writing the SCB pointer into the SCB FIFO. The sequencer then takes the oldest entry off this FIFO, processes it and when the command is complete writes the SCB pointer into the outgoing SCB FIFO (while optionally signaling an interrupt).

The scratch RAM is used by the sequencer microcode as temporary storage, it is not supported by the simulation model.

5.2.2 Configuration

files: *IO/ahc.c*
IO/ahc.h

The following table summarizes the runtime parameters for the Adaptec SCSI controller model.

Parameter	Description	Default
AHC_scbs	Number of on-chip SCB entries	32

Table 22: Adaptec SCSI Controller Parameters

Currently the only configurable parameter is the number of on-chip SCB entries. This number should be set reasonably since the adapter model does not support SCB paging, hence it must be able to store all pending requests in its on-chip SCBs.

5.2.3 Functional Description

A SCSI command is sent to the controller by filling in an available SCB entry. This is normally accomplished by setting the SCB pointer to the desired entry and clearing the *SCB-cnt* register. This register supports an auto-increment mode, in which the byte offset is incremented after every access. The device driver then writes the SCB data into the current SCB using a sequence of uncached byte or word writes. When the SCB is set up, the SCB pointer is written into the SCB FIFO. The sequencer continuously polls this FIFO, and if it is not empty it removes the first entry and starts processing the request. It reads the SCSI command with parameters via DMA from main memory, then reads the scatter/gather DMA vector if one is used, sends the command to the SCSI device and waits for a response. If the device accepts the request, the controller starts the DMA data transfer on both the SCSI and the host bus. Note that all data transfers are described by a scatter/gather vector, which is a series of physical address and length pairs. When the transfer is complete, the adapter writes the SCB pointer in the outgoing SCB FIFO and interrupts the host processor if the *command-complete* interrupt bit is set. If the device disconnects from the adapter during the transfer, the adapter records the current scatter/gather segment number and current address/length in the SCB and resumes polling the incoming SCB FIFO. If, while the adapter is either polling the FIFO or reading the SCSI command or DMA vector, a device issues a reconnect command, the adapter searches for the SCB that corresponds to the request. An SCB matches the request if the target and LUN number match the SCB, as well as the queue tag in case of a queued command. Also, the adapter allows only one non-queued request per target. It keeps a list of pending non-queued requests and stalls the incoming SCB FIFO if a second non-queued request for the same target is waiting.

5.2.4 Data Buffering, Flow Control and SCB Implementation

files: *IO/ahc.c*
IO/ahc.h

In the AIC7770, all data transfers between the host bus and SCSI bus pass through the data FIFOs. Control registers determine when the adapter issues the next DMA transaction on the host bus (based on full or empty ratio of the FIFO). Since the SCSI bus does not model data transfers at a fine granularity, this detail can not be modeled accurately. Instead, once the device has accepted the request (or after a reconnect), the adapter starts issuing DMA requests to the system bus interface at a steady rate determined by the SCSI transfer rate and the per-request data block size. For instance, if the SCSI bus can transfer 2 bytes per clock (100 ns period), the adapter issues a 64 byte cache line write request every 3.2 µs. For writes to main memory, the adapter uses a slightly lower rate since it must first receive the data from the SCSI bus before it can write it to memory. However, depending on system bus contention, this simple DMA model may lead to some asynchronicity, e.g. a SCSI write request may be completed before the adapter has read all the data from memory. To avoid data loss in these cases, the adapter calls the SCSI completion routine only at the beginning or end of the DMA transfer, irrespective of the SCSI bus timing.

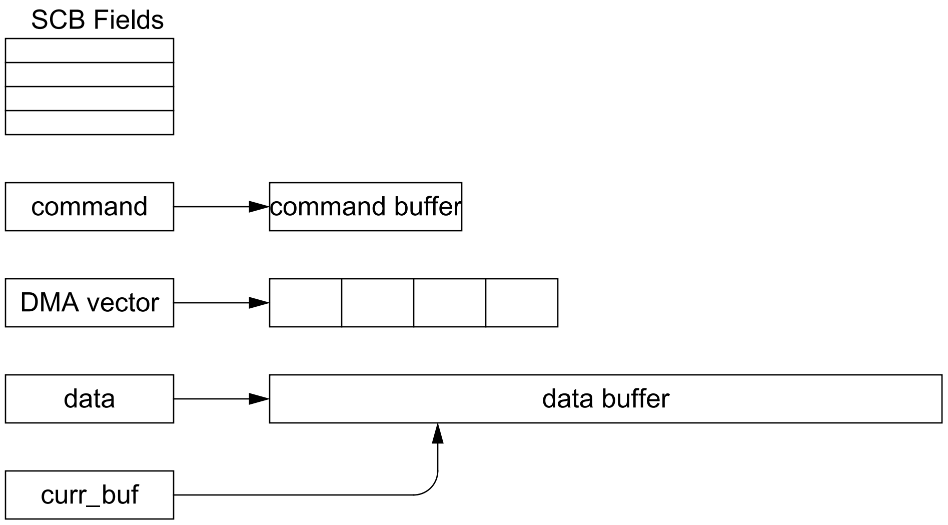

Figure 15: Internal SCB Structure

Each SCB structure is augmented with pointers to buffers which are used during the transfer. The command buffer is allocated before the SCSI command is read by the adapter. If the command indicates that data is to be transferred, the adapter allocates a buffer for the DMA scatter/gather vector and reads the vector into the buffer. Note that the adapter model reads the entire vector before starting the actual data transfer, rather then reading the vector one element at a time. Finally, the adapter allocates a buffer to hold the entire amount of data to be transferred. All buffers are freed when the entire transfer is complete.

5.2.5 Host Interface

files: IO/ahc.c
IO/ahc.h

The host interface consists of two callback functions that handle read and write requests to the adapter and a routine that is used to issue bus transactions. Since the control registers are byte-addressed, the read and write callback routines process each request at a byte granularity. For every request, these routines scan through the affected addresses, performing any necessary side-effects. For instance, when the write callback routine detects a write to the SCB FIFO, it reads the value that was just written, inserts it into the SCB FIFO and adjusts the FIFO status. Similarly, upon a write to the clear-interrupt register it resets the internal interrupt status and clears the interrupt register. Whenever the SCB pointer is changed, the routine copies the new SCB array entry into the SCB registers. For all writes to an SCB register, the routine also updates the SCB array entry to which the current SCB pointer points. Upon a write to the SCB input FIFO, or when the sequencer halt bit is cleared, the routine also schedules the sequencer event.

The SCSI adapter model uses two routines to implement the host bus master. In addition, each SCB maintains the current physical address, a pointer to the current buffer location, a flag indicating whether to read or write main memory, a length counter and a completed-length counter. The length counter indicates how many more data transfer request have to be issued, it is decremented every time a request is sent to the system interface. The completed-length counter indicates how many more request need to be completed, it is used by the sequencer to wait for completion of outstanding requests before finishing a transfer and deallocating the buffers.

The routine *ahc_dma* allocates a new system bus request structure, fills in the physical address based on the SCBs current address and the request length (always a L2 cache block). Each host bus request allows for some source-module specific data and two pointers to functions that are called when the request completes. The SCSI adapter uses this to store a pointer to the current buffer, the request length (based on current address alignment and remaining length) and a pointer to the SCB in the request. It then subtracts the current request length from the remaining length, increments the address and buffer pointer accordingly and calls *SCSI_cntl_host_issue* to send the request to the system bus interface. The completion routine *ahc_perform* looks up the physical address in the simulator page table, copies the specified amount of data between the buffer and physical memory and decrements the remaining completed-length counter.

As a PCI device, the SCSI adapter needs to provide an address space mapping routine. The generic SCSI adapter forwards mapping requests to the specific adapter because different adapters will have different address space requirements, which are best dealt with in the adapter specific code. The routine *ahc_pci_map* takes as arguments a pointer to the SCSI controller structure, the value written to the PCI configuration register, the node and bus module ID of the controller, the function and register number with the devices configuration space and pointers to size and flag variables. If the value that was written is -1, the routine returns the required address space size in the size pointer, otherwise it installs a mapping for the configuration registers using the new address in the page table and registers the address space with the address map.

5.2.6 Sequencer Implementation

files: IO/ahc.c
IO/ahc.h

The sequencer is the main control part of the SCSI adapter, both in the real system and in the simulator. It is responsible for polling the SCB input queue, reading SCSI commands and DMA vectors via the DMA engine, transferring data between the host and SCSI bus and for handling SCSI reconnect requests.

In the simulator, the sequencer is implemented by the *ahc_sequencer* event handler. Normally, the event is inactive. The event gets scheduled when the host processor writes to the SCB input queue or when a SCSI device issues a reconnect request to the adapter. Furthermore, the device driver can start and stop the sequencer explicitly by setting a bit in the host-control register. Consequently, the event is also scheduled when the sequencer is started. The following figure shows a state diagram of the sequencer.

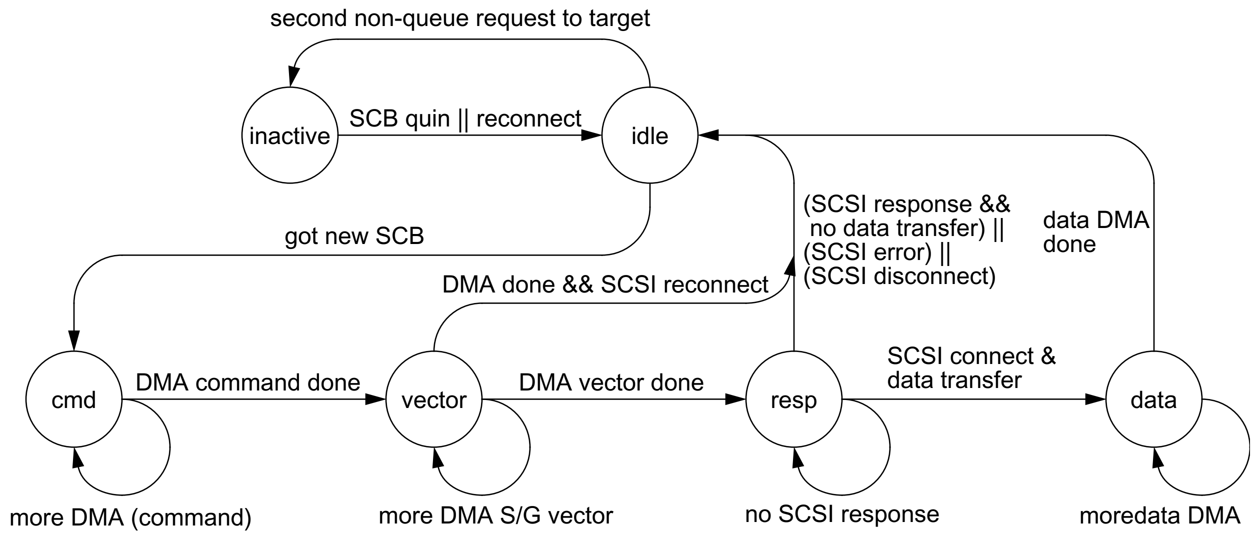

Figure 16: Adaptec SCSI Adapter Sequencer State Diagram

Note that the inactive state corresponds to when the sequencer event is not scheduled. When the sequencer is activated, it first checks if a reconnect request has arrived at the adapter. If this is the case, it picks the SCB that corresponds to the pending request and continues processing it, otherwise, it checks the SCB input queue for new requests. If a request is taken from the input queue, the sequencer checks if this is a second non-queued request to a target, in which case the request must be stalled. If the sequencer has found a valid request, it moves to the command state. In this state, the sequencer allocates the command buffer and initiates the DMA read of the SCSI command into this buffer. If the buffer has already been allocated, the request has received a SCSI reconnect response and the sequencer does not need to read the command again, otherwise it starts issuing host bus transactions until the remaining DMA length is zero. It then waits until the required amount of data has been transferred and moves to the *dma_vector* state.

This state is similar to the command state, except that the sequencer reads the DMA scatter/gather vector. Again, if the buffer has already been allocated or the DMA vector length is 0 it skips this state, otherwise it issues a series of host bus transactions. Once the sequencer has read the SCSI command and DMA vector, it is ready to arbitrate for the SCSI bus and issue the request to the target. However, if in the meantime a *reconnect* response has arrived, the sequencer drops the current request, returns to the *idle* state and picks up the *reconnect-request*. Otherwise, it issues the current request on the SCSI bus and removes it from the SCB input queue. Note that this is most likely handled differently in the real system, since the real SCSI bus allows the adapter to stall even after it has been granted the bus if it needs to wait for the DMA vector to be read, whereas in the simulator this is not possible and the request can only be issued (and removed from the queue) after the command and vector have been read. At this point the sequencer also modifies the *current-SCB* register to point to the SCB that is currently being processed.

In the *response* state, the sequencer waits for the SCSI response. If the response is *connect,* the device has accepted the request and is starting the data transfer. If the current request requires a data transfer, the sequencer advances to the data state and starts transferring data on the system bus. If the request does not require a data transfer, the sequencer completes the request by pushing it onto the SCB output queue and optionally interrupting the host processor. It then returns to the *idle* state where it looks for another request. Similarly, if a *timeout* response is received, the sequencer interrupts the host CPU, but does not push the request onto the output queue. If a *complete* response is received, the request does not require any data transfer and can be completed immediately. Finally, if the adapter receives a *busy* response, the device is currently unable to process the request, in which case the sequencer puts the request back into the input queue and returns to the idle state.

In the *data* state, the sequencer issues host bus transactions while incrementing the physical address and decrementing the remaining length. Whenever it reaches the end of a DMA segment it advances the current segment pointer and continues at the new physical start address. Note that the *connect* or *reconect* response indicates the amount of data to be transferred, which may be less than the total amount of data for this request. If at the end of the transfer the adapter receives a *complete* response, it finishes the request by moving it to the output queue and interrupting the host processor, otherwise it simply returns to the *idle* state. In the latter case, the target will issue a *reconnect* request and the sequencer will continue the data transfer where it left off.

The sequencer uses three different intervals when it is polling. During command and vector DMA as well as when waiting for the response, it uses a fast polling interval of 2 system bus cycles. When it is writing data via DMA it uses an interval that roughly corresponds to the SCSI bus transfer rate, that is it attempts to issue host bus transactions at the same rate as the SCSI bus can provide the data. For DMA read transfers, it uses a slightly faster rate since it must be able to keep the SCSI bus fully utilized. Note that these are only simulator abstractions, since the SCSI bus data transfer is not actually simulated at a byte level. The adapter model actually performs the SCSI request only at the beginning of the data transfer (for reads) or at the end of the data transfer (for writes), regardless of the SCSI bus timing. If the host bus is particularly congested, it may happen that a SCSI write is complete on the SCSI bus while the adapter is still reading data from main memory.

5.2.7 Statistics

files: IO/ahc.c
IO/ahc.h

The SCSI controller gathers various statistics during the simulation. Among these are interrupt handler latency, per-request latencies and a request summary. Interrupt latency is measured separately for command-complete interrupts and SCSI interrupts (timeout etc.). For each interrupt the model measures the latency from the time when the interrupt is sent to the bus interface (essentially when the event occurred) until the device driver reads the interrupt status register, as well as the time until the device driver clears the interrupt. The simulation model measures the minimum, maximum and average latencies and reports absolute times based on the specified core clock frequency.

In addition, the SCSI controller records various latencies for each request it handles. When a new request is written into the SCB input FIFO, the controller records the current simulation time as start time in the SCB. The start time is used to measure input queue latency, connect-latency (latency until the SCSI target accepts the request), completion latency (latency until the request is complete and enters the SCB output queue) and total request latency (latency until the request is removed from the SCB output queue).

6 SCSI Disk

This component models a simplified SCSI disk. It accurately models the mechanical behavior of the drive, the effects of caching, prefetching and delayed writes. Data is made persistent by storing it in an external data file.

6.1 Configuration

files: IO/scsi_disk.c
IO/scsi_disk.h

Various features of the SCSI disk model can be configured at simulator startup time. The following table summarizes the configuration parameters.

Parameter	Description	Default
DISK_name	text description of drive	IBM Ultrastar 9LP; 9 GB
DISK_seek_single	single-track seek time in milliseconds	0.7 ms
DISK_seek_average	average seek time in milliseconds	6.5 ms
DISK_seek_full	full media seek time in milliseconds	14.0 ms
DISK_seek_method	method for computing seek times (disk_seek_none, disk_seek_const, disk_seek_line, disk_seek_curve)	disk_seek_curve
DISK_write_settle	write settle time in milliseconds	1.3 ms
DISK_head_switch	head switch time in milliseconds	0.85 ms
DISK_cntl_ov	controller overhead in SCSI bus cycles	20 cycles
DISK_rpm	rotational speed	7200
DISK_cylinders	number of cylinders on media	8420
DISK_heads	number of disk heads	10
DISK_sectors	number of sectors per track	209
DISK_cylinder_skew	skew between adjecent cylinders	30 sectors
DISK_track_skew	skew between tracks	50 sectors
DISK_request_queue	size of request queue	32
DISK_response_queue	size of response queue	32
DISK_cache_size	onboard cache size in KBytes	1024 KByte
DISK_cache_segments	number of cache segments	16
DISK_cache_write_segments	number of cache write segments	2
DISK_prefetch	enable disk-side prefetching after reads	1
DISK_fast_write	enable fast writes to cache	0

Table 23: SCSI Disk Configuration Parameters

Parameter	Description	Default
DISK_buffer_full	buffer full ratio for reconnect after read	0.75
DISK_buffer_empty	buffer empty ratio for reconnect after write	0.75
DISK_storage_prefix	path to disk storage files (index and data)	empty

Table 23: SCSI Disk Configuration Parameters

The default parameters represent the IBM Ultrastar 9ES drive with 4.5 GByte capacity. The routine *SCSI_disk_init* initializes the necessary data structures, reads the parameters from the configuration file and calls *SCSI_device_init* to register the device with the SCSI bus.

6.2 Request Handling

files: IO/scsi_disk.c, IO/scsi_bus.c
IO/scsi_disk.h, IO/scsi_bus.h

6.2.1 Incoming Requests

When a request arrives for the disk drive, the bus model calls the routine *DISK_request*. This routine checks if the request parameters are legal and responds with *reject* if any of the requested block numbers is less than 0 or greater than the number of blocks in the drive.

A non-queued request is rejected with status *busy* if the queue is not empty or the drive is not idle, unless the current request is a read-induced prefetch or a cache writeback. Similarly, a queued request is rejected if the input queue is full. After the new request has passed these tests, it can be put into the queue for processing. Note that this is done even for non-queued requests since these are only accepted if the queue is empty.

In the absence of a full read cache hit, the drive aborts read requests with status *disconnect* and queues the request. Similarly, all other requests that may require mechanical disk activity are queued before the drive disconnects. If the requests immediate-flag is set, the drive can return a *complete* status at this point, otherwise it disconnects and reports completion later.

Since write requests involve an immediate data transfer, the drive attempts to attach a write cache segment to the request. If none is available, the request is rejected with status *busy*. If the total transferred data fits in the available segments, the drive returns the status *complete* if fast writes are allowed, or *disconnect* otherwise. Finally, if the request size exceeds the available write cache space, the drive transfers the maximum data size possible and returns *save_data_pointer*.

When receiving an *inquiry* request, the disk drive returns various model information in a structure that is pointed to by the request buf-field. This structure contains the device type (0 - disk), flags that indicate that the drive is capable of queueing requests and supports 16 and 32 bit wide SCSI busses, and the textual description of the drive model. Similarly, the *read_capacity* request is handled by filling in a structure with the number of blocks on the device and the block size (512 bytes).

The disk model maintains a sense-data structure for every possible initiator on the bus (e.g. 16 structures on a 16 bit bus). When a request encountered an error such as an invalid block number, the sense-data structure for the initiator is set up with the relevant information. Normally, the initiator will request *sense-data* immediately after it detected the error. The data structure is cleared when an error-free request arrives.

6.2.2 Input Queue Handling

For requests that may require access to the media the routine schedules the request-handler event, unless the disk is not idle or the event has already been scheduled. In the first case, the drive will schedule the event when the current request is completed.

Further request handling is performed by three event handler routines. The *request_handler* routine takes requests off the input queue, determines any partial or full cache hits and initiates disk seek operations. First, the routine checks if the request queue is empty. If this is the case and the disk cache contains a buffered write, it creates a write request which transfers the data from the disk cache to the media.

For a *read* or *prefetch* request, the routine checks if the request hits in the cache. If the current request is a read-induced prefetch (a completed read converted to a prefetch) and the input queue is not empty, the prefetch is aborted and the *request_handler* event is rescheduled. In case of a full hit, a reconnect-transaction with status *complete* is issued. In case of a partial cache hit or cache miss, the request is attached to a cache segment (the segment that contains the last data sector that hit in the cache, or a new segment) and a disk seek operation is initiated.

For a *seek* request, the routine simply initiates a seek operation.

If the current request is a *write*, it is first committed, which allows subsequent reads to detect a hit on the write segment. If the disk allows fast writes, the immediate flag is set and all data has been transferred, the drive may now start the writeback if the input queue is idle, otherwise it discards the request and reschedules the *request_handler* event. If fast writes are not allowed, the drive initiates a seek operation to the first sector to be written. If no write segment is available, the disk can either accept the request and disconnect (if the immediate flag is set in the request), or return a *busy* status.

In case of a *sync-cache* request, the drive checks if the cache contains any dirty segments. If this is not the case and the immediate-flag was not set, completion is reported by issuing a *reconnect* request with status *complete*, otherwise the request is simply discarded. If the cache contains dirty segments, a seek operation is initiated to the first dirty sector.

Finally, for a format-request with no immediate flag, the drive reports completion at this time without any other operation.

6.2.3 Disk Seek Operation

When initiating a seek operation to a specific sector, the disk drive determines the delay based on current head, cylinder and rotational angle, and schedules the *seek_handler* event for the time when the head arrives at the target sector. Before initiating the seek operation, the disk checks if a seek is already in progress (due to a cache writeback or read-induced prefetch). If this is the case, the current seek is aborted and the disk model computes the current cylinder based on the seek start time, total seek distance and seek direction. Note that since the total seek time includes the rotational delay, the head may already be at the target cylinder even though the seek delay has not passed.

The seek event handler is called when a seek operation is complete. If the current request is a seek with the immediate flag not set this routine reports completion of the request and reschedules the *request_handler* event. Otherwise it schedules the *sector_handler* event to occur when the current sector has been transferred.

6.2.4 Sector Transfer

Whenever a sector has passed under the head, the *sector_handler* event is triggered. If the current request is a read or prefetch, the routine inserts the current sector into the cache segment that is attached to the current request. If the current request is a read-induced prefetch and the cache segment is full, or the input queue is not empty, the prefetch is aborted. If a read or true prefetch request fills up a cache segment, the request is attached to a new segment.

The buffer full ratio determines how much data must be read into the buffer before the drive reconnects to the initiator to start transferring data. If the current transfer size exceeds this ratio, the event handler estimates the transfer time for the remaining sectors. If this transfer time is less than the time it takes to transfer the entire data over the bus, it issues a *reconnect* request with status *complete*, otherwise the status is *save_data_pointer*. This approach is necessary because the data transfer is simulated as two discrete events and the duration as well as the completion status must be known before the end-of-transfer event is scheduled. Normally, the drive would simply start transferring data and disconnect when the buffer is empty.

If the current request is a write and an entire cache segment has been transferred, this segment is marked as clean. If the more data needs to be transferred from the initiator because the write segments did not provide enough space for the entire request, a *reconnect* transaction is issued with status *complete* or *save_data_pointer*. The routine then finds the next write segment in the cache. If no more writes are pending, it reschedules the *request_handler* event.

For a *sync_cache* request, the event handler routine first checks if the entire segment has been written. If this is the case, the segment can be marked as clean and a new dirty write segment is attached to the request. If no segment is found and the requests immediate-flag is not set, completion is reported at this point.

After performing request-specific operations, the routine either reschedules the *sector_handler* event if more data needs to be transferred and the next sector is on the current track, or performs another seek operation. If, however, all sectors have been transferred, the routine can either convert a read request into a prefetch if the input queue is empty or report completion of a synchronous write (immediate-flag not set). It then reschedules the *request_handler* event.

6.3 On-board Cache

files: *IO/disk_mech.c*
IO/scsi_disk.h

In most disk drives the on-board cache is divided into a small number of cache lines, or segments. In addition, only a subset of these segments can be used for write requests. The model does not restrict writes to any specific segments, it only limits the number of write segments. Unlike hardware caches, disk cache segments can start at arbitrary sectors and contain variable numbers of sectors.

When a read request is processed, the drive first checks for a partial or full read hit. A partial hit is currently only detected if the first N sectors hit in the cache. In this case, the request is attached to the segment that contains the highest matching sector number, the additional sectors that are read to fulfil the request are appended to this segment.

Writing data occurs in two phases. When the request arrives and is enqueued, data is buffered in any of the write segments. In order to preserve correct ordering, read requests to not detect matches to these segments until the write has reached the head of the input queue and is committed. The disk cache is modeled as an array of segment descriptors, each containing the start and end sector number, an LRU counter, a write flag and a write-commit flag.

The routine *DISK_cache_getsegment* is used to attach a request to a segment. For read requests, it searches through all read segments plus any committed write segments and searches for the segment that contains the highest numbered sector in the block number range provided as argument. This search may also find a segment that ends at the lower bound of the block number range, in which case this is not a cache hot but the newly read sectors can be appended to this segment. For write requests, the routine searches only through the committed write segments for an overlapping segment. If no segment is found, the routine returns a new segment number.

The routine *DISK_cache_hit* checks if the specified block number range is in the disk cache, and if it is a partial or full hit. It searches for the segment containing the current *start_block*. If that segment contains the entire block range, a full cache hit is detected, otherwise *start_block* and length are adjusted and the search continues until either segment contains the remaining blocks (full hit), or only the first blocks (partial hit, adjust and repeat).

The routine *DISK_cache_insert* adds blocks to a specific cache segment. It detects any partial overlaps and either overwrites the entire segment or appends to the end of the segment. It returns the number sectors actually written, as the segment might not provide enough space for all the blocks. This routine is used for prefetch and read requests when data is read from the media.

Similarly, the routine *DISK_cache_write* inserts write data into the cache. However, it first checks if the current number of write segments is already at the maximum. If this is not the case, it allocates a new segment, inserts the data and designates the segment as a non-committed write segment. This is repeated until all data is written or no more write segments can be allocated. For delayed write operations, write segments are found by calling *DISK_cache_get_wsegment*, which returns the number, start block and length of the first committed write segment. A write request is committed by the routine *DISK_cache_commit_write*. It first invalidates all overlapping read segments and then marks all write segments that contain data within the specified sector range as committed. Writes are completed when all data within a segment has been written to the disk, the routine *DISK_cache_complete_write* clears the write bit of the specified segment so that subsequent reads can use the data to satisfy requests.

6.4 Disk Mechanics

files: IO/disk_mech.c
IO/scsi_disk.h

The mechanical components of the disk drive are modeled by a routine that compute at which sector a disk head is at a given time, a routine that computes the time to position the head over a specific track and a routine that computes rotational delays. The disk is assumed to start rotating instantaneously at time 0. The sector under a head at a given time can be computed by calculating the rotational angle at that time and converting it into a sector number.

The disk arm movement delay requires the most complex modelling since it is a non-linear process. During every movement, the disk arm first accelerates to its maximum speed, coasts for some time and then decelerates. Depending on the seek distance, the coast phase might not happen, and the disk arm might not even accelerate to its maximum speed. For a write, the arm takes additional time to fine-position the head. Switching the active head also requires some time. Although this can be hidden for longer seeks, it might dominate the seek time for short distances.

Disk arm movement time can be modeled in many different ways, the simulation model supports four different methods. The simplest method assumes instantaneous arm movements. The second method models a constant seek time, as specified in the average seek time parameter. A more accurate algorithm models the seek time as a three-point line, where the first third of the distance corresponds to the average seek time, and the remaining distance is a fraction of the full seek time. The most accurate algorithm models seek times as a three-point fitted curve. The total time to seek to a given sector is computed as the sum of the arm movement time to reach the target track, and the rotational time to reach the desired sector, plus any additional write-settle time.

The routine *DISK_do_seek* performs the actual seek operation by first computing the target cylinder and head, which are then used to calculate the arm movement delay. The routine then calculates which sector will be under the head when the arm movement is complete and adds the rotation delay to reach the target sector. It then schedules the *seek-event* to occur when the target sector has been reached, sets the disk state to seek and updates the current cylinder and head variables.

6.5 Persistent Storage

files: *IO/disk_storage.c*
IO/scsi_disk.h

Persistent data storage in the disk model is achieved by writing all disk blocks to a file on the simulation host system. To reduce the size of this file, only modified (written at least once) disk blocks are saved, and a hash table is used to translate logical block numbers to offsets within the data file. The hash table is also saved in a corresponding index file. Index and data files are created in the current directory, unless a different location is specified with the *DISK_storage_prefix* configuration parameter. The filenames are formed from the node number, SCSI bus number and device number of the respective disk module. For instance, the file *disk_01_00_08.idx* is the index file for a disk with the SCSI device ID 8, attached to SCSI bus 0 on node 1.

The hash table is an array of 1024 entries, each containing a logical to physical block mapping and a pointer to the next element. Each entry in the array is the head of a simple linked list of descriptors.

The routine *DISK_storage_init* initializes the storage data structure. It initializes the hash table and forms the filenames. It then checks if the files already exist and creates them if necessary. Upon creation, an empty hash table is written to the index file. The routine then reads the hash table from the index file into the internal data structure. In the index file, each linked list of descriptors is stored sequentially, a NULL next-pointer indicates the end of the current list.

The routine *DISK_storage_read* reads blocks from the data file into a buffer provided as argument. It opens the data file, iterates through the logical block numbers and searches the hash table for each block. If a hash table entry was found (the block has been written before), it reads the block from the data file into the buffer. If no hash table entry was found, the routine returns a block of zeroes. In the end, it closes the data file to reduce the risk of corruption due to simulator crashes.

The routine *DISK_write_storage* writes blocks from the buffer into the data file. Similarly to the read routine, it iterates through the logical blocks. If a matching hash table entry was found, it overwrites the block in the data file. However, if no hash table entry was found, it appends a new descriptor to the appropriate linked list, appends the block to the data file and updates the new entry with the correct block mapping. At the end, it closes the data file and writes the entire hash table to the index file. Note that if opening either index or data file fails, the read and write routines print a warning and retry after a delay of several seconds. This approach avoids aborting the simulation in the case that the disk files are only temporarily unavailable because the file server is down. To preserve persistent storage space, new blocks consisting only of zeros are not written to the data file nor is the sector appended to the hash table, as the read routine will return zeros by default if the sector is not found.

7 Platform Specific Details

Currently, ML-RSIM runs on SPARC/Solaris and Mips/Irix hosts as well as Intel x86 based Linux hosts. While the simulator always interprets SPARC binaries and thus shields the executables from the host platform, several simulator components are dependent on the specific platform.

7.1 Memory References

files: Processor/endian_swap.h

The simulated SPARC instruction set specifies big-endian addressing of memory locations. If the simulator is running on a little-endian host such as an Intel x86 based system, the mismatch between the simulated binaries assumption about data layout and the hosts native format must be corrected by the simulator.

In ML-RSIM, main memory and all memory-mapped locations are addressed big-endian, while all arithmetic is performed using the hosts native format. Consequently, an endian swap may be required whenever a bus module accesses main memory or memory-mapped locations. This includes all load and store instructions as well as all device-side modifications to memory-mapped device registers and some simulator traps that directly access simulated memory regions. If the simulation host is a big-endian system, the endian swap routines are empty and cause no overhead.

7.2 Simulator Traps

files: Processor/traps.cc

Simulator traps are used by the simulated Lamix operating system to request services from the underlying host operating system, for instance when mirroring native files and directories into the simulated filesystem. Since Lamix is Solaris compatible, all simulator trap arguments can be passed from Lamix to an underlying Solaris OS without modifications. However, if the simulation host is not running Solaris, trap arguments and must be translated between the Lamix/Solaris definition and the native format. This applies to all structures such as *flock*, *dirent*, *msg* and *stat* as well as the values of constants such as *AF_UNIX*, *F_FREESP* and *SI_SYSNAME*, among many others. In addition to the required format and value conversion, an endian swap may be required if simulated memory is accessed directly.

7.3 Floating Point Arithmetic and Exceptions

files: Processor/funcs.cc, Processor/signal_handler.cc, Processor/fsr.cc, Processor/linuxfp.c Processor/fsr.h, Processor/linuxfp.h

The simulator emulates floating point exceptions by setting the hosts floating point control word to reflect the simulated processors FSR register before performing a floating point instruction, and then catching any FP signals and recording them in a global variable. Various hosts require different formats and access methods for these control registers. In addition, the actual signal handling process may differ from host to host.

Part V: The Lamix Kernel

The ML-RSIM runtime system is a fairly complete Unix-compatibel kernel. It provides most features of Unix including process control, signals, file systems and shared memory. These functions are mainly based on NetBSD and to a lesser extend Linux source code. Currently, only the memory management subsystem is simplistic, it does not support paging or even sophisticated page allocation policies. The current version of Lamix contains a significant portion of NetBSD source code from version 1.4C, which is an intermediate release following 1.4.3 (the last official 1.4 release), and before 1.5. The NetBSD sources are largely unmodified, only when necessary variable names or memory-management related subroutine calls have been adjusted.

The kernel is a standalone executable that is linked at the virtual address 0x8000 0000. The simulator loads the file into memory upon startup and starts executing at the entry point that is specified in the ELF header.

Upon startup, the following steps are performed:

1. set up page table for init process and map kernel text and data
2. enable MMU
3. setup kernel data structures
4. detect and configure hardware devices
5. configure pseudo devices
6. mount root filesystem
7. mount filesystems on attached disks
8. open standard input, output and error files
9. fork user process(es)
10. parent process (init):
    change credentials of new process to that of the user running the simulation
    call sys_wait() until only one process is left, then close all remaining open files
11. child process:
    call sys_exec() to start application
12. wait for all children to finish execution
13. unmount file systems
14. execute any shutdown hooks

1 Overview

1.1 Compiling and Linking ML-RSIM Applications

files: apps/makefile_default

The only requirements for ML-RSIM applications are that they need to be linked statically, be compiled and linked in 32-bit mode and that the executable file is in 32-bit ELF format. The default *makefile* in the *apps* subdirectory sets up the correct compiler and linker flags to accomplish this. The *makefile* can be included from an application *makefile*. It assumes that the following directories exist:

- src/ - source files
- obj/ - object code
- execs/ - executable
- outputs/ - optional directory for output files

The top-level makefile should define the variables *SRC* and *TARGET* appropriately.

However, applications can also be compiled without the default makefile, it is only necessary to specify the *-xarch=v8* flag for compilation (to produce 32 bit code) and linking (to link with 32 bit system calls) and to specify the *-dn* flag for static linking.

1.2 Memory Layout

files: mm/init.s, mm/misc.s, kernel/process.c
mm/mm.h

The following figure shows the virtual memory as seen by a process.

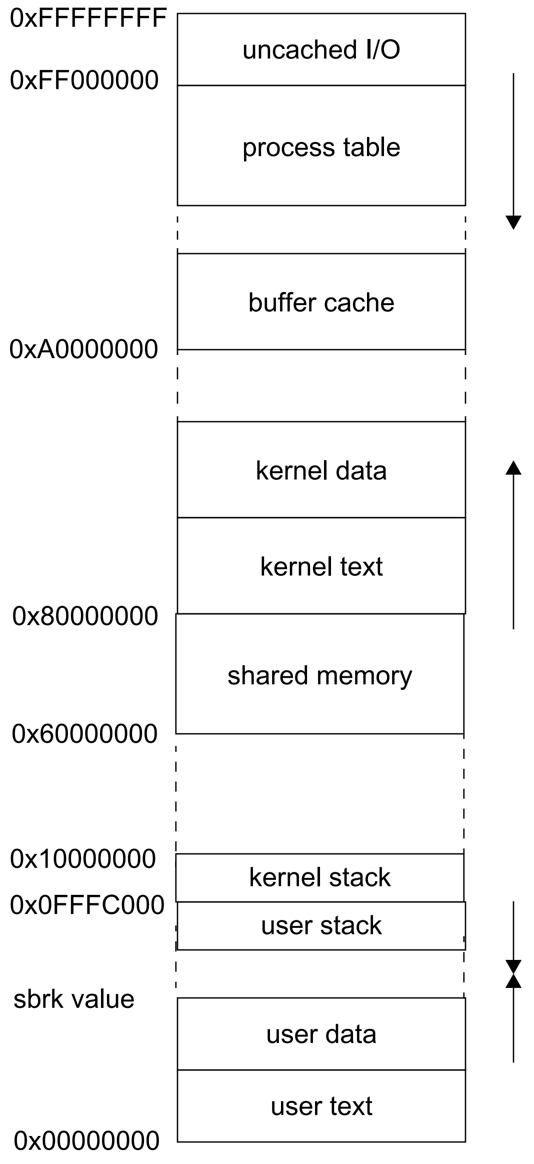

Figure 17: Per-Process Virtual Memory Layout

Each process virtual address space starts at address 0 and spans a total of 256 MByte. The kernel stack segment starts at virtual address 0x10000000, it has a fixed size of three pages followed by an unmapped guard page. The user stack starts below the kernel stack guard page, it's initial size is one page and it may be extended at runtime up to a maximum size of 1 MByte. (Currently, the stack size is set to 0.5 MB and the stack does not grow dynamically, problems with the interaction of window overflow exceptions and page faults required to map the entire stack when the process is created.)

The physical address space is shared among all the processes in a node. Generally, the upper 1 MByte (defined in *mm/mm.h*) is used for memory-mapped I/O devices. The lower 1.5 GByte are divided equally among up to 6 user processes. Each process may use up to 256 MByte of virtual and physical memory. However, the total number of processes can be changed before the kernel is compiled by setting constants on *mm/mm.h*, with a corresponding adjustment in virtual address space size. The segment between 0x60000000 and 0x80000000 is reserved for shared pages. The remaining physical memory starting at 0x80000000 and extending to the I/O segment is used for kernel text, data, page tables and other system purposes.

The following figure shows an example of a 2-process node configuration.

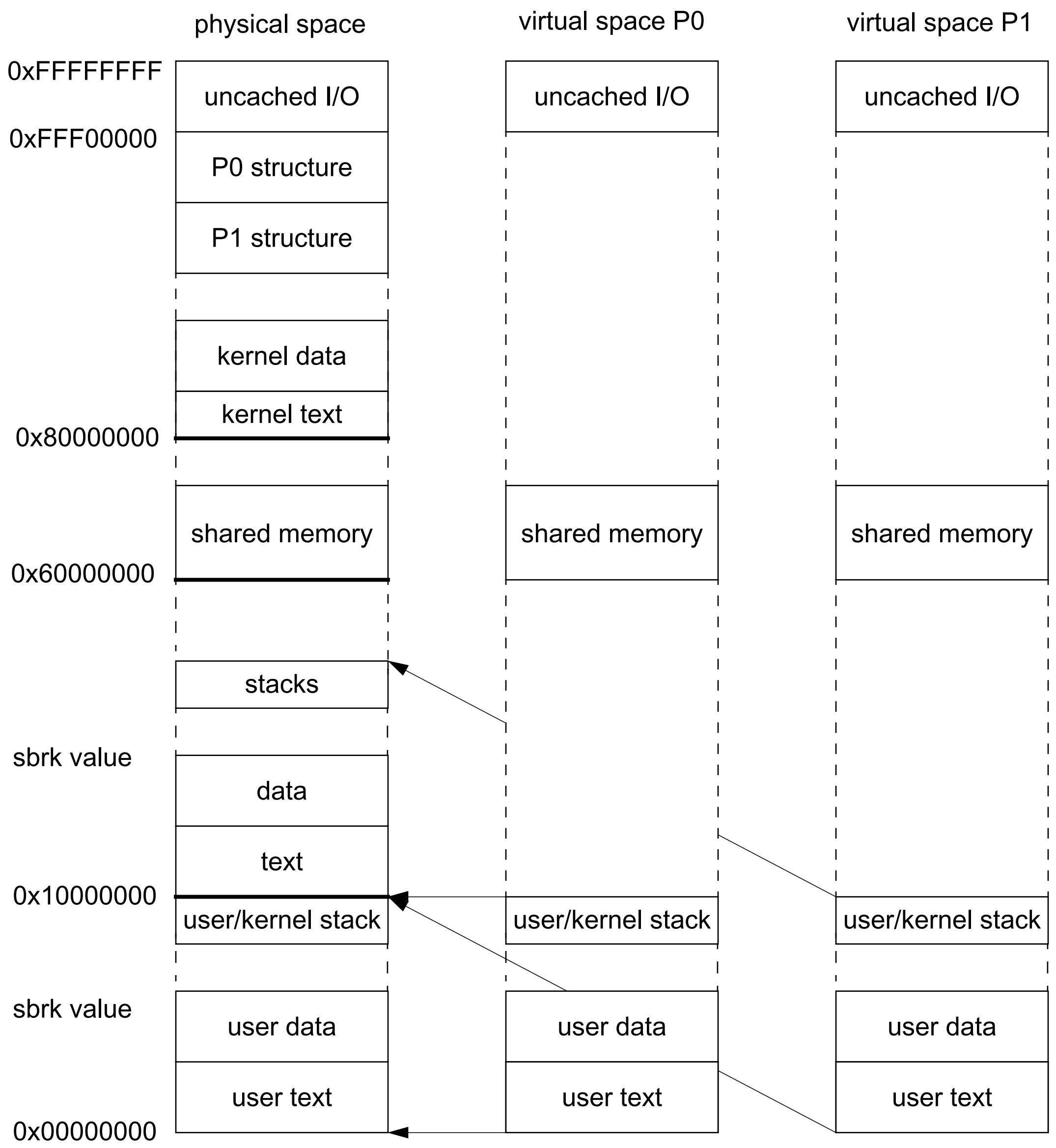

Figure 18: Physical Memory Layout for 2 Processes

The space starting below the I/O segment is used to hold the process table. Each process has 1 MByte available for its process structure and page table. The highest page is used as the process structure, the next lower page is that processes top-level page table, while the bottom-level page tables are allocated below.

1.3 Page Table Organization

files: mm/init.s, mm/tlb_trap.s, mm/misc.s, mm/segv.s
mm/mm.h

Address translations are stored in a simple direct-mapped 2-level page table. The first level is indexed by the 10 most significant bits. Each 32 bit entry contains a page-aligned pointer to the level-1 page table and a valid bit (bit 0). If the entry is valid (bit 0 is set), the entry points to a page that contains the actual translations for this address. The base address of this level-0 page is stored in the processors *tlb_context* register.

Each level-1 page is indexed by bits 21:12 of the virtual address. Its 32 bit entries contain the physical page number in bits 31:12 and various access attributes in the lower bits, in the same format as the data portion of a TLB entry.

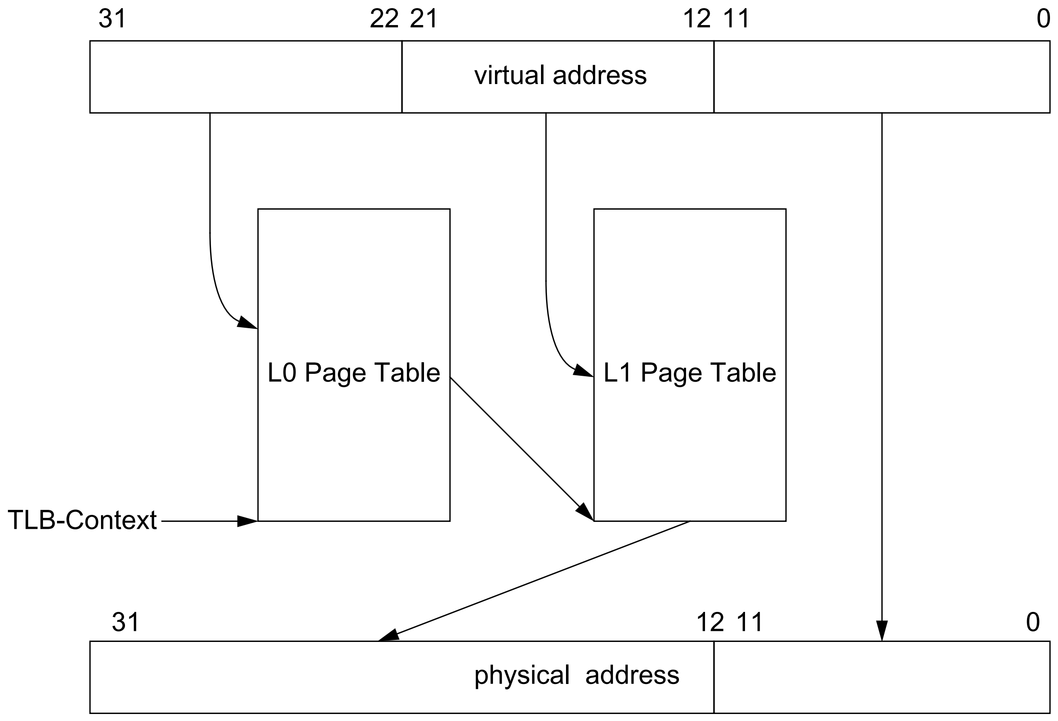

Figure 19: Page Table Organization

1.4 Process Structures

files: *kernel/process.c*
kernel/process.h

The kernel maintains a process control structure for each active process. The following table summarizes the elements of this control structure:

Field	Description
state	process state (running, ready, blocked ..)
ticks	number of clock ticks the process has been running
flags	various control flags
pid	process ID
pgid	process group ID
sid	session ID
uid	user ID
gid	group ID
ngroups	number of groups the process belongs to
groups[16]	lits of groups the process belongs to
parent	pointer to parent process structure
next_p, prev_p	pointer to next and previous process
next, prev	pointer to next and previous process in varios queues
children	head pointer for list of children
next_s, prev_s	pointer to next and previous sibling
phys_low	physical start address of process
pt_low	current lower bound of page table
text_low, text_high	boundaries of text-segment, exclusive text_high
data_low, data_high	boundaries of data segment, exclusive data_high
stack_low, stack_high	boundaries of stack segment
onfault	continuation adress upon address fault (used by copyin/copyout)
global_r[8]	space for global registers when inactive
in_r[8]	space for input registers when inactive
y_r	space for Y registers when inactive
context_r	space for context (page table pointer) register
pil_r	space for processor interrupt level register

Table 24: Process Control Structure

Field	Description
cwp_r	current window pointer when inactive
fp_r[32]	floating point registers when inactive
fprs_r	floating point control register when inactive
kernel_sp	kernel stack pointer
exit_code	exit code (upper byte) or exit signal number (lower byte)
rtime	process runtime in seconds and microseconds
uticks	CPU time used in user mode in clock ticks
sticks	CPU time used in system mode in clock ticks
signal	bit vector of pending signals
signal_mask	bit vector of blocked signals
signal_oldmask	signal mask saved during sigsuspend system call
signal_flags	signal handling flags
signal_action[32]	array of signal dispositions, incl. flags and mask
sig_stack_sp	alternate signal stack pointer
sig_stack_size	alternate signal stack size
sig_stack_flags	alternate signal stack control flags
signal_info[32]	info structures for pending signals
wait_channel	address of kernel structure process is waiting on
wait_mesg	string describing reason process is blocked
timeout	timeout value for wait
it_real_value	realtime timer expiration (in ticks)
it_prof_value	profile timer expiration value (in ticks)
it_virt_value	virtual (user) timer expiration value (in ticks)
it_real_incr	realtime timer refresh value
it_prof_incr	profile timer refresh value
it_virt_incr	virtual timer refresh value
it_real_timer	pointer to active timer descriptor
it_prof_curr	profile timer current value
it_virt_curr	virtual timer current value
text	pointer to executable file vnode
filedesc	list of open file descriptors
cwdinfo	current working directory and root directory

Table 24: Process Control Structure

Field	Description
dupfd	help variable for dup system calle
stats	resource usage statistics for this process and its children
limits	process resource limits
stat_io_latency	I/O latency statistics handle
stat_io_copy	I/O copy overhead statistics handle
stat_idle	process idle time statistics handle

Table 24: Process Control Structure

The *next_p/prev_p* pointers are used to form a doubly-linked list of processes maintained by the kernel. Similarly, the *next/prev* pointers can form a doubly-linked list of runnable processes, or blocked processes waiting for a resource. Processes are also linked in a list of siblings which starts from the parents children pointer. The remaining variables keep track of the virtual address boundaries of the process, they are used when the data segment or stack needs to grow. The process structure also includes variables to control signal delivery, an array of pending signal informations, timer information and a fixe-size array of open file descriptors.

The process control structures are located in each process' system segment, along with the page table. Below the process structure is the page table base (top-level page table) located. The following figure shows the layout of the entire per-process data structures.

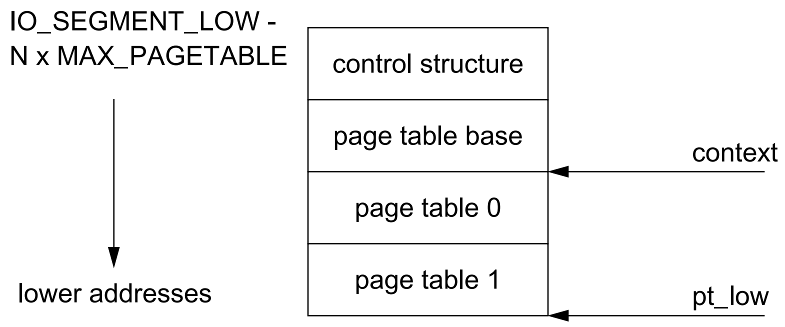

Figure 20: Process Table Entry Layout

Currently, the process structures and page tables are statically allocated. The first process's segment is located below the I/O space, the second process's segment would be at *IO_SEGMENT_LOW - MAX_PAGETABLE* and so on. A bit map indicates which process structure slots are allocated or free.

2 Startup and Initialization

files: mm/init.s, kernel/kernel.c, kernel/process.c

When the simulator starts executing, it has already loaded the kernel code and data, allocated physical pages for the kernel stack segment and initialized the stack pointer. The processor is in kernel-mode, address translation and interrupts are disabled.

First, the startup code in init.s initializes the trap table base address register (*%tba*), and determines the TLB type. The default trap table contains TLB miss handlers for fully-associative TLBs. If the system uses direct-mapped or set-associative TLBs, the startup code copies the corresponding handler codes into the trap table.

Following the trap table setup, the routine installs page table mappings for the kernel text, data and stack segment. The startup routine then places a trap-frame on the kernel stack that is used by the first process to switch from the initial kernel mode to user mode. This trap frame contains only cleared registers, *TState* is set to user mode with TLBs and interrupts enabled. The *sys_execve* routine which is called by the initial process to load applications modifies the return PC and stack pointer in this stack frame to jump to user mode and start executing.

After enabling the MMU, the startup code calls *kernel_init* to initialize various kernel data structures as well as I/O devices and additional interrupt handlers. This routine first finishes setting up the initial process, initializes further kernel data structures such as the timers, shared memory descriptors and file descriptors. It then performs the device autoconfiguration and calls initialization routines for the file systems, attaches pseudo devices and finally mounts the root filesystem. The *init* process then opens the stdin/stdout/stderr files and sets up the current working directory and root directory. At this point the system is ready to execute user processes. If a batch file was specified, the *init* process reads it and either forks the required number of processes if a concurrent batch was requested, or forks only the first application process for sequential batch jobs. For each forked child it changes the credentials of the new process to the credentials of the user running the simulation, so that the process has proper access to the user files on the host system. In case of a parallel batch command, the *init* process simply waits for the number of active processes to be reduced to one and then terminates the simulation. For a sequential batch job, it repeatedly forks one new process and waits for its completion. Note that kernel threads must not be counted towards active processes as they may never execute and may not exit on their own. When the *init* process is the only remaining process, it closes the input and output files, shuts down the virtual file system layer which includes synchronizing all disks and unmounting the file systems, calls any shutdown hooks installed by device drivers and halts the simulation.

Each forked child process calls *sys_execve* to load the respective application and then returns. When *kernel_init* was called by the startup code, the return address was modified to point to the system call return code. When the child returns, it will execute this code which restores the processor state from the trap frame that was placed on the kernel stack, switches to user mode and starts executing the application.

2.1 Init Process Setup

files: *mm/init.s, mm/misc.s*
mm/mm.h

The assembler routine *creat_init* is responsible for bootstrapping the system by allocating memory and installing page table mappings for the kernel data structures. The routine allocates and clears memory for the top-level page table and maps the kernel text and data pages. Note that this involves allocating a page for the level-1 page table. The routine then allocates memory and installs the page table mappings for a kernel stack and a process structure, and sets the stack pointer to the kernel stack. At this point, the necessary data structures are setup to call C functions (which may need a stack) and to map further pages (need page table pointers in the process).

2.2 Idle Processes

files: *kernel/kernel.c, kernel/schedule.c*
kernel/kernel.h, kernel/process.h, kernel/cpu.h

Since the simulator OS was originally intended to support shared-memory multiprocessors, it was designed to provide a separate idle-thread for each processor. An idle-thread is a normal process, except that it does not have user-space text, data or stack pages mapped. The *kernel_idler_init* routine is called during kernel setup. For each processor, it allocates a process descriptor and the associated page table structures. Note that the process descriptor is not taken from the pool of general user-process structures, but is allocated from a separate pool. Initially, the return address is set to point to the *idler* routine.

An idle-thread simply spins on the head of the run-queue and the active process count. If the number of active processes is less than one, the idle-thread halts, effectively stopping simulation for this CPU. If the run-queue is not empty, it calls the context-switch routine. Note that due to race conditions, the run-queue might be empty by the time the idle CPU attempts to switch to the process. In this case, the context switch routine returns to the idle thread. Each idle-thread records the time that it enters and exits the idle loop and adds the idle-time to the total idle-time in a per-CPU structure.

2.3 SMP Startup (not enabled yet)

files: *kernel/kernel.c, kernel/schedule.c*
kernel/kernel.h, kernel/cpu.h

This section discusses aspects of the original design that are still present in the current kernel but are not used. The current version of Lamix does not support multiple processors !

The kernel maintains per-CPU information in an array of cache-line aligned structures. Each of these structures contains pointers to the current process and to that CPUs idle process, a *timespec* structure that accumulates the idle times, a *need_reschedule* flag and the local interrupt count used for synchronization.

If the simulated system has more than one CPU, only the first processor performs the kernel initialization. The remaining processors spin on a flag that is set when the kernel setup is complete enough to switch to the idle threads. The flag is set by the *kernel_init* routine after the first process has been forked, but before the *init* process blocks in *sys_wait*. When the other processors detect that the flag is set, they initialize the *tlb_context* register with a pointer to the init-processes page table, enable the TLBs and load the pointer to the idle process from the CPU-specific structure.

2.4 Kernel Parameters

files: kernel/kernel.c
kernel/kernel.h

The simulator passes all command line parameters that follow the *-F* executable flag to the kernel. Normally, the kernel assumes that the first command parameter is the user process and the remaining parameters are passed to this process. During initialization, the kernel scans the list of arguments until it reaches the first argument that does not start with a '-' (dash). These arguments are considered kernel arguments, they can be used to pass special configuration information to the kernel. The following table summaries the supported kernel arguments.

Parameter	Description	Default
-root	run user process as root (useful when running system administration tools)	off
-nomount	do not mount simulator disk partitions during startup	on (mount disks)
-mtasync	mount filesystems in asynchronous mode	off
-bufcache=N	percentage of physical memory used for buffer cache	200 KB + 5% of main memory
-bufpages=N	number of pages used for buffer cache	none
-bufs	number of buffer structures in buffer cache	none
-input=<file>	file from which to read as standard input	/dev/null
-output=<file>	file to which to write standard output and standard error messages	<executable>.std-out <executable>.stderr
-batch	read commands and arguments from specified file and execute them sequentially	none
-parbatch	read commands and arguments from specified file and execute them in parallel (multiprogrammed)	none
-cwd=<dir>	change to specified directory before starting user processes	none
-noclock	disable clock interrupt handler	off
-ccdconf=<file>	name of concatenated device configuration file	ccd.conf
-lfs_cleanerd=<file>	pathname of LFS cleaner daemon	lamix/lfs_cleaner

Table 25: Kernel Parameters

The first two options are useful when running system administration tools that must be run as root, or that access a disk device directly. The asynchronous filesystem options mounts all file systems on simulated disks in asynchronous mode, which means that metadata updates are not immediately written back to disk. This feature improves write performance significantly at the expense of a higher risk of filesystem inconsistencies in case of a system crash. The option should only be used when simulation time is to be minimized, since by default BSD file systems are mounted in synchronous mode.

The *-bufcache* option can be used to specify the percentage of main memory that is to be used for the buffer cache. By default, the system allocates 10 percent of the first 2 MB (= 200 KB) and 5 percent of the remaining physical memory for the buffer cache. The next two options (*-bufpages* and *-bufs*) explicitly specify how many pages should be used for the buffer cache and how many buffer structures should be used. When not specified, the number of buffer structures is equal to the number of buffer pages.

The input file option is equivalent to the input redirection on a normal Unix command line (e.g. cmd < file). The output file name option redirects both standard output and standard error to an alternative file. By default, two different files are created by the simulator, with the name being derived from the name of the simulated executable or the subject name specified at the command line. This option overwrites the directory and subject name specified at the simulator command line (options *-D* and *-S*).

The batch file parameters cause the kernel to read the commands and parameters from a text file instead of from the command line. In this case the first simulator command line parameters is assumed to be the batch file name. Each line in the text file contains a user command including full (non-relative) path and all necessary commands. Empty lines or lines starting with '#' are ignored. Comments following a command are not allowed. When the *-batch* option is used, the kernel executes one command after the other. The simulator statistics are reset at the beginning of each command and a statistics summary is written after each command completes executing. In parallel mode (*-parbatch*), the kernel forks and executes all specified commands at once and then waits for completion of all commands before writing the statistics summary.

The *-noclock* option disables clock interrupts until the last user process terminates, thus eliminating the cost of clock interrupt handling. This option should not be used for processes that rely on timing information such as clock system calls, alarm signals or other timeout mechanisms. Disk activity, however, is still possible. To facilitate proper disk synchronization and filesystem unmounting, the kernel enables clock interrupts when the last process terminates.

The command line argument and environment vectors are made available to other kernel subsystem through global variables. This allows device drivers and other modules to search for command line arguments specific to that subsystem. Two such arguments are the ccdconfig file and the LFS cleaner daemon path.

2.5 Kernel Memory Allocation

files: *kernel/malloc.c*

Kernel memory allocation is currently very simple. The kernel uses the native C-library for many utility routines, including memory management. However, the original *_sbrk* routine has been replaced by a routine that grows or shrinks the kernel data segment by installing or removing page table entries. The variable *kmem_high* keeps track of the upper bound of the kernel data segment and is used to detect a possible overrun of the file cache structures.

Several memory management routines have been equipped with wrappers that raise the interrupt priority to a save level before calling the original routine. Kernel routines can use *kmalloc* to allocate memory, *kfree* to return it to the free memory pool, *krealloc* to change the size of an allocated memory portion, and *kmemalign* to allocate an aligned memory chunk. Arguments to these routines are identical to the original routines.

2.6 Kernel Patching

files: *machine/patch.c*
machine/patch.h

The machine-dependent subsystem provides two routines that provide a limited patch facility for the memory image of the kernel. These routines rewrite subroutine calls and jumps to new target addresses and are intended to allow device drivers to install modified versions of core kernel routines, if absolutely necessary. Occasionally, a device driver requires a hook into certain memory management routines, or process related calls. In such cases, it may provide its own version of such routines, which may even call the original routine after or before performing its specific tasks. However, since this method requires intimate knowledge of the core kernel functionality, it should be reserved for rare circumstances.

The *patch_call* routine searches for the first occurrence of a call to a specific subroutine, starting at a specific address, and replaces it with a call to a new subroutine. Upon success, it returns the address of the patched instruction, otherwise it returns 0.

The *patch_sethi_jmpl* routine searches for a pair of *sethi/jmpl* instructions to a certain address, while starting the search at a specified address. It then replaces the target address with a new address, and returns the address of the *sethi* instruction upon success.

3 Processes

3.1 Process Creation

files: kernel/process.c, kernel/fork.c, kernel/signals.c, kernel/misc.s, mm/misc.s
kernel/process.h, kernel/signals.h, mm/mm.h

Creating a new process involves creating and initializing the process structure and copying the parent process page table mappings and other attributes. The routine *create_process* first finds an unused process structure in the array of available process structures starting below the I/O space. A bit mask protected by a lock is used to indicate which structures are in use. The routine then maps the new process structure and initializes various elements such as the process ID and timer variables.

Various routines are used to either copy process attributes from the parent to the child process, or to initialize child process attributes. The routine *copy_pagetable* installs page table mappings for the new process which are a copy of the parent processes page table entries. It maps the text, data and stack segments at the same virtual addresses but with a different physical mapping, determined by the *phys_low* variable of the new process. The shared segment mappings are copied at the bottom level of the page table, that means that for every bottom level page table entry of the parent an equivalent entry is created for the child. Finally, the kernel segment is a shared segment, the top level page tables of all process point to the same set of bottom level pages. In this way, modifications in the kernel segment mappings are visible to all processes.

The routine *copy_memory* is responsible for physically copying the parents segments to the child. It copies the entire text segment, data segment and stack segment (both user and kernel), using physical addresses. During this process, address translation and interrupts are turned off. Before copying the stack segments, the routine flushes the register windows so that their contents can be copied properly.

Process attributes such as user ID and process group ID are copied by the routine *copy_attributes*. Similarly, *copy_signals* copies the signal dispositions from the parent to the child process, and *copy_fd* copies the open file descriptors (while incrementing the reference count) and current working directory/root directory.

3.2 Process Termination

files: kernel/exit.c, kernel/wait.c, kernel/misc.s
kernel/process.h

Processes may terminate normally by calling the *exit* system call, or abnormally due to a signal. The latter case is currently not implemented, fatal errors such as a segmentation fault cause the entire simulation to halt.

The exit system call first wakes up the parent process if it is waiting for the child, as indicated by the *PWAIT* flag in the child processes flags field. It then clears the pending signal mask, blocks all signals, removes the child from any timer queues and closes all open files as well as the current working directory and root directory. At this point, the user memory (text, data, user stack) can be released and unmapped, followed by unmapping any shared memory segments (currently not supported). The process is now removed from the process list, its state changed to *zombie* and the exit status recorded. If the process exited due to a normal *exit* system call, the exit status is stored in the upper byte and the lower byte is zero. If the process terminates due to a signal, the lower byte indicates the signal number and the upper byte is 0.

If the terminating process has any children left, it wakes up the *init* process and moves its children to the *init* processes children list. If its parent process has the *NOCLDWAIT* flag set, it indicates that it does not intend to collect the exit status of its children, and the process itself becomes a child of the *init* process. Finally, the process sends a *SIGCHLD* signal to its parent and performs a final context switch.

The exit status of processes can be collected by two different system calls. The *wait* call blocks the caller until one of its children has terminated or changed state. The system call scans the list of children of the calling process, checking each in turn if it is in *zombie* status or if it has changed state from stopped to runnable or vice versa. In the first case, the system call reaps the process structure and returns the process ID in register *%o0* and the exit code in register *%o1*. If a process has changed state, the process ID is returned in register *%o0* and the nature of the state change is indicated in *%o1*. If the caller has no children, the system call returns with the *ECHILD* error code. If the process has children but none has terminated or changed state, the system call blocks on the callers process structure.

A more flexible interface is provided by the *waitsys* system call. It takes as arguments a process ID and an ID type specifier, a pointer to a *siginfo* structure and a set of flags. The ID may be a process ID type, or process group ID, a session ID, user ID or group ID. The caller can specify if it wants to collect status of terminated children or stopped children, and if it wants to block if no status can be returned. The system call scans the callers list of children until a process is found whose identity matches the specified ID. If this process has terminated and the caller has specified the *WEXITED* flags, the *siginfo* structure is filled with information such as the exit status, process ID, *utime* and *stime* of the child. If the *WNOWAIT* flags is not set, the child process is now reaped. The system call returns 0 on success. If a stopped child is found, the *siginfo* structure is filled with similar information and the system call returns. If no child was found that satisfies the search criteria, the routine returns the *ECHILD* error code, otherwise it blocks unless the *WNOWAIT* flags was set by the caller.

Reaping a process involves removing the process from the parents list of children, adding the *utime* and *stime* values to the parents *cutime* and *cstime* values, unmapping the process structure and returning the process structure to the pool of free structures.

3.3 Kernel Threads

files: kernel/kthread.c
kernel/kthread.h

Kernel threads are regular processes, except that they share virtual memory with the *init* process. Hence, kernel thread overhead is almost identical to regular processes, except that thread creation and termination overhead is lower due to shared address space. The routine *kthread_create* creates a kernel thread. Its structure is similar to the *fork* system call, except that it takes additional arguments that specify the start address of the thread and a thread argument. Kernel threads do not support signals and share virtual memory with the *init* process. As a result, only a subset of the process structure needs to be initialized or copied from the *init* process.

In addition to the normal kernel thread creation, the kernel supports a queue of deferred kernel threads. This queue is in fact a simple linked list that contains pointers to arbitrary functions and one argument per entry. The kernel calls each function in turn after the device drivers have been configured and the file systems have been initialized. This list allows device drivers to delay the creation of kernel threads until after the system is reaching a stable state. However, the general nature of the list allows the delayed execution of many other initialization tasks.

3.4 Process Statistics

files: kernel/process.h

Each process structure contains several user statistics handles that, when set to a value not equal -1, are used to record the number and duration of various process activities. Currently, the kernel records the frequency and latency of all I/O system calls, the duration of each I/O copy routine and the frequency and duration of idle periods for each process.

3.5 Program Execution

files: kernel/exec.c, kernel/exec_elf32.c
kernel/exec_elf.h

The *execve* system call overlays the current process with a new executable. The new executable inherits all credentials and attributes of the original process, except that files that have the *close-on-exec* flag set are closed and signals that are caught are reset to the default disposition.

First, the system call checks if the executable file exists and the calling process is allowed to executed it. It then copies the argument and environment lists to a scratch area, since these lists reside in an address space that is going to be overwritten. The scratch area is allocated starting at the upper boundary of the data segment. Next, the calling processes text and data segment is unmapped. If the executable resides on the *HostFS* filesystem, the instruction cache is flushed. This is necessary because file transfers from the *HostFS* filesystem are not performed via DMA. After this, the system call copies the argument and environment lists onto the user stack (while growing the stack if necessary) and copies the signal trampoline code at the top of the user stack (in case it has been corrupted by the calling process).

Currently, only executables in 32-bit ELF format can be loaded. The routine *exec_elf32* first reads the ELF header, which includes the entry point and the number and location of program and section headers. The routine then reads each program header and installs page table mappings for every loadable program segment. Finally, it reads the section headers and either copies data from the file to the appropriate virtual address (text or initialized data), or clears the corresponding memory area.

After the new program has been loaded, the system call modifies the trap frame by clearing all registers, setting the trap-PC to the entry point and setting the stack pointer to the new window restore area. In addition, the window restore area on the user stack is modified so that the stack pointer points to the beginning of the argument list.

The system call then calls *closeexec_fd* to close all files that have the *close-on-exec* flag set, and *execsigs* to reset the signal dispositions of signals that are caught by the original process to the default. Finally, if the parent is waiting for a status change (*PWAIT* flag is set), the parent is woken up.

3.6 Context Switch

files: kernel/misc.s

The context switch routine takes the pointer to the new process structure as argument. It first saves the current process state into that processes control structure. This involves saving the input registers (including stack pointer and return address), current window pointer and the Y register. The *FPRS* register indicates if the FPU is enabled and if the lower or upper half of the floating point registers have been modified. If this is the case, the routine saves the FP registers as well.

At this point, the routine flushes the register windows and changes the *current_proc* pointer to the new process, and restores the new process state (input register, FP registers, FPRS, Y, CWP). It then loads the page table base address of the new process into the *tlb_context* register and flushes the TLB and restores the interrupt level to the value provided as second argument. This is necessary since the context switch routine executes with all interrupts disabled. A special version of this routine is called by the *exit* system call. It does not save any state of the current process, since that process has terminated.

Two higher-level routines are responsible for voluntary and involuntary context switching. The routine *switch_process* takes the first process off the run-queue and switches to it. If the run queue is empty, the routine selects the idle-process for the current CPU. The idle-process spins on the run-queue (without locking it) until it is not empty, or until no processes are active in the system. When the queue becomes non-empty, the idle-process calls the context switch routine again. When a new process is chosen, the tick value in its process structure is set to the maximum number of ticks per time slice (10). At every clock tick, this value is decremented. If it reaches zero, the interrupt handler sets the *need_resched* flag for the current CPU. This flag is checked upon every return from kernel to user mode, if it is set a context switch is initiated by calling *round_robin*. This routine enters the current process at the end of the run queue and then calls *switch_process* to choose a new process to run (this might be the same process if no other process is ready).

The Lamix kernel provides a facility that allows other subsystems to register context switch callback routines. These routines are called before the actual context switch, and take the old and new process pointer, the CPU that performs the context switch, and a callback-specific pointer as arguments. The kernel maintains a list of such callback routines, each element consisting of a pointer to the actual routine plus a void pointer argument. Initially, no callback routines are called during context switches. However, when the first callback routine is registered, the calls to the low-level context switch routines are patched and redirected to a routine that first traverses the list of registered callbacks, calling each in turn, before calling the original context switch code. Applying a patch at runtime allows kernel configurations that do not require such callbacks to avoid the overhead of traversing an empty list.

3.7 Synchronization

files: kernel/sleep.c, kernel/synch.s
kernel/process.h, kernel.sleep.h, kernel/sync.h

Currently, the kernel synchronization does not support multiple processors, since this would require careful placement of locks to protect kernel data structures beyond what is necessary on a uniprocessor system. The routines *splxxx* (where xxx refers to an interrupt priority) are called when the kernel needs to disable interrupts below a certain priority level. For instance, *splbio* disables all interrupts that deal with block I/O devices, in order to protect the buffer cache. The *splxxx* routines return the original priority level which should be restored at the end of the critical section by calling *splx*.

Process-level synchronization is handled by the *ltsleep* routine. It takes as arguments a wait channel (the address of a resource to wait on), a wait message that identifies the reason why the process is blocked, a timeout value and an optional simple lock. The kernel implements sleep queues as a hash table of linked lists, hashed by the wait channel. When a process calls sleep, the wait channel and wait message are recorded in its process structure and the process is linked in the appropriate sleep queue. If the timeout value is not zero, a timer is started that will wake up the process after the specified interval. The caller can also specify if it wants to be woken up when a signal arrives. If this is the case, the *SINTR* flag is set in the process structure. If a signal is currently waiting, the routine immediately removes the process from the sleep queue, stops the timer and returns. If an interlock is specified, the lock is released while the interrupt priority is raised to avoid race conditions. Similarly, if a NULL pointer was specified as wait channel, the routine returns. Otherwise, the process state is changed to *SLEEP* and a context switch initiated by calling *switch_process*. When the original process is woken up, it continues executing at the same point. The *TIMEOUT* flag in its process structure indicates if it was woken up by the time out routine. In this case, the sleep routine returns the *EWOULDBLOCK* error code. Otherwise, the timeout counter is stopped. If the process intended to be woken up by signals, and the signal set is not empty, the routine returns the *EINTR* error code, otherwise it returns 0. If an interlock is specified, the lock is re-acquired before the interrupt priority is lowered. The interlock feature allows processes to block while holding a lock without risking deadlocks.

Blocked processes are woken up by the routine *wakeup*. This routine takes a wait channel as argument. It searches through all processes in the appropriate sleep queue and compares the wait channel with its argument. For every process that needs to be woken up, it removes the process from the wait queue, changes its state to *READY*, enters it in the ready queue and sets the *need_reschedule* flag. Note that this wakes up all processes that where blocked waiting for a resource. If this resource can not be shared, all but one process will return to sleeping. This is controlled at a higher level, for instance by the file lock manager. When a blocked process times out, it is removed from the sleep queue, its state changes to *READY* and it is entered into the run queue.

4 Exception Handlers

files: interrupts/traptable.s

The kernel trap table contains exception handler entry points for all exceptions. The handler address is computed by adding the exception code shifted to the left by 5 bits to the trap table base address (register *tba*). Each trap table entry provides space for 8 instructions. The exception codes of performance critical trap vectors such as TLB misses and window traps are set such that the following trap table entries are unused, thus providing space for up to 32 instructions.

The trap table is located in unmapped address space. When taking an exception, the processor disables the instruction TLB (instruction fetches use physical address), and disables the data TLB for TLB miss exceptions. Most exception handlers re-enable the I-TLB and branch to a routine in mapped address space. Window traps and TLB miss handlers are inlined into the trap table.

4.1 TLB Miss Handler

files: interrupts/traptable.s, mm/tlb_trap.s
mm/mm.h, interrupts/traps.h

When a TLB miss occurs in the processor, the *tlb_bad_addr* register contains the faulting address, and the *tlb_tag* register contains the tag for the entry that needs to be inserted. In addition, in case of a direct-mapped or set-associative TLB, the *tlb_index* register points to the index that needs to be filled.

The TLB miss handler first reads the context register, which points to the top-level page table for the current process, and the faulting address. The 12 most significant bits of the address are the index into the top-level page table. After loading the top-level entry, the handler checks if the entry is valid (bit 0 is set).

If this is not the case, it installs an invalid mapping in the TLB, which means that the entry is marked valid, but the mapping-valid bit is cleared. When the trapping instruction re-executes, it will trigger a TLB fault exception.

If, on the other hand, the top-level entry is valid, the handler uses bits 19 to 10 as index into the level-1 page table. It loads the corresponding level-1 entry and writes it into the TLB data register. Writing the data register has the side-effect of writing the TLB entry that the *tlb_index* register points to. Note that this entry might be invalid, but the handler does not need to check this. An invalid entry will trigger a TLB fault exception.

The TLB miss handler for the fully-associative TLB performs essentially the same operations, but in addition reads the *tlb_random* register and writes its contents into the *tlb_index* register, in order to implement the random replacement strategy.

By default, the trap table contains the inlined code for fully-associative TLBs. If the processor uses set-associative or direct-mapped TLBs, the startup routine copies the appropriate code into the trap table.

4.2 Generic Trap Handling

files: interrupts/traps.h

External interrupts and system calls use the generic trap entry and exit code defined as a macro in *interrupts/traps.h*. The trap entry code first needs to determine if the interrupted process was executing in kernel or user mode, by checking the privileged-bit in the *tstate* register. If the process was executing in user-mode, the interrupt handler must change the stack pointer to the kernel stack. The kernel stack location can be determined by reading the *kernel_sp* element in the current process structure, pointed to by the *current_proc[cpu_id]* array.

After switching to the kernel stack, or if the process was already executing in kernel mode, the interrupt handler allocates a new stack frame that provides space for the registers that need to be saved as well as for a register window that might need to be saved later on. It then saves the trap-registers (*tstate, tpc, tnpc*), the *y* and *pil* register, allocates a new register window and saves the input registers on the stack. Note that these store instructions might incur TLB misses and the TLB miss handler overwrites registers *%g1* through *%g4*. Subsequently, the interrupt handler switches back to the normal set of global registers and saves them on the stack as well. Since the trap register have been saved, and in order to allow arbitrarily deep nesting of interrupts, the handler now decrements the trap-level, which acts as an index into the trap-register stack. It then changes the *pil* register to the maximum value, effectively enabling all interrupts, sets the interrupt-bit in the processor status register, and loads the trap type into register *%o0* since it is an argument for the interrupt handler routine.

The trap-exit code performs almost the exact opposite of the entry code. However, before restoring state it checks if the process has any pending signals, in which case it calls *do_signal* with the process pointer, signal set and a pointer to the saved registers as arguments. An additional argument indicates if the handler will return with the *Retry* or *Done* instruction, as this affects the signal trampoline code. If no signal is pending, or after the signal has been handled, the exit code checks if the *need_reschedule* flag is set. If it is, it calls *round_robin* to perform a context switch. Note that these two checks are only performed when the exception/trap handler returns to user mode. This is always the case for system call handlers, but needs to be checked for all other interrupt and exception handlers.

The trap exit code now switches to the normal globals (in case the interrupt handler used the alternate set) and restores the global registers from the stack. It then switches to the alternate globals and disables interrupts. Before restoring the trap-registers, it needs to increment the trap level. Finally, the handler restores the input registers and restores the register window.

4.3 Memory Protection Fault Handler

files: interrupts/traps.s, interrupts/traphandlers.c, interrupts/traptable.s
mm/mm.h, interrupts/traps.h

The memory protection fault handler routines are called whenever an application attempts to access an illegal address. The instruction fault handler first checks if the fault happened in user or kernel mode. If the processor was running in kernel mode, the error is considered unrecoverable, a message is printed and the system halts. If the system was running in user mode, the routine prints diagnostic information and sends a *SIGSEGV* signal with the proper signal information to the current process. Unless caught, this signal will cause the process to abort immediately.

The data fault handler also first checks if the system was running in kernel mode, and halts the system if this was the case. If, on the other hand, the system was in user mode, it checks if the faulting virtual address lies within the stack segment. In this case, the handler routine grows the stack by allocating physical pages and installing the corresponding mappings in the page table. The stack is grown towards lower addresses until the faulting address is reached. Note that for every installed mapping, any matching TLB entries must be flushed by probing the TLB and writing an invalid entry in its place.

If the faulting address is not in the process stack segment, and the *onfault* variable in the process structure is set, the data fault routine modifies the trap return address and returns. This feature is used by several routines that copy data to or from user space. Rather than validating user addresses explicitly, these routines set the *onfault* variable of the current process, such that in case of a protection violation, the trap handler would resume execution at a code segment that returns an error code to the caller.

If the faulting address is neither a stack address nor a the *onfault* variable is set, the trap routine prints a diagnostic message and sends a *SIGSEGV* signal to the current process.

Note that if the data fault occurs in a window overflow trap, the exception handler code first ‘manually’ saves the window onto kernel stack and adjusts the window control registers before it jumps to the C routine. When restoring state, it performs the opposite steps, restoring the window and adjusting the window control registers before returning to the trapped instruction.

4.4 Bus Error

files: interrupts/traps.s, interrupts/traphandlers.c, interrupts/traptable.s, kernel/misc.s interrupts/traps.h

A bus error exception is triggered when a memory access is misaligned according to the SPARC architecture definition. Normally, this condition is an error, except for double and quad floating-point loads and stores. These instructions need only be word aligned. However, the processor triggers a bus error trap even for these cases. Hence, the bus error trap handler first checks if the faulting instruction is word aligned and is a double or quad floating-point load or store. If this is the case, it emulates the instruction and returns. Floating point loads are emulated by copying the source memory location byte by byte into a local variable (which is guaranteed to be aligned) and then loading the aligned variable into the destination register (routines *set_fpreg_xx*). Stores are emulated by first storing the register value into the local variable and then copying it byte by byte into the destination memory location.

If the instruction is truly misaligned and is not emulated, and the system was in kernel mode, the trap handler prints a diagnostic message and halts, otherwise it sends a *SIGBUS* signal to the current process and returns.

4.5 Miscellaneous Error Traps

files: *interrupts/traps.s, interrupts/traphandlers.c, interrupts/traptable.s interrupts/traps.h*

Several trap handlers exist to deal with exceptions such as illegal opcodes, privileged instruction or division by 0. Generally, these handlers first check if the system was running in kernel mode when the exception occurred. If this is the case, a diagnostic message is printed and the system halts. Otherwise, a similar message is printed and the exception handler sends the appropriate signal including detailed signal information if applicable to the current process. The following table summarizes the various exceptions and corresponding signals with signal information.

Exception	Signal	Signal information

illegal trap	SIGILL	code = ILL_ILLTRAP
		fault.addr = pc
		fault.trapno = tt (trap type)
		fault.pc = pc

bus error	SIGBUS	code = BUS_ADRALN
		fault.addr = faultaddress
		fault.trapno = tt (trap type)
		fault.pc = pc

illegal instruction	SIGILL	code = ILL_ILLOPC
		fault.addr = pc
		fault.trapno = tt (trap type)
		fault.pc = pc

privileged instruction	SIGILL	code = ILL_PRVOPC
		fault.addr = pc
		fault.trapno = tt (trap type)
		fault.pc = pc

FP disabled	SIGILL	code = ILL_ILLOPC
		fault.addr = pc
		fault.trapno = tt (trap type)
		fault.pc = pc

FP error	SIGFPE	code depends on FSR register
		fault.addr = pc
		fault.trapno = tt (trap type)
		fault.pc = pc

division by 0	SIGFPE	code = FPE_INTDIV
		fault.addr = pc
		fault.trapno = tt (trap type)
		fault.pc = pc

Table 26: Kernel Parameters

Exception	Signal	Signal information

instruction fault	SIGSEGV	code = SEGV_MAPERR / SEGV_ACCERR
		fault.addr = pc (= fault_addr)
		fault.trapno = tt (trap type)
		fault.pc = pc

data fault	SIGSEGV	code = SEGV_MAPERR / SEGV_ACCERR
		fault.addr = fault_addr
		fault.trapno = tt (trap type)
		fault.pc = pc

Table 26: Kernel Parameters

The FP error signal code depends on the value of the FSR register, in particular on the exception type and in case of an IEEE floating point exception also on the subtype. In case of an instruction or data fault, the signal code specifies whether the fault is due to an invalid mapping or an access to a privileged address.

4.6 System Trap Handler

files: syscall/syscall.s, interrupts/traphandlers.c
interrupts/traps.h

As described earlier, the system call interface expects the system call number in register *%g1* and all arguments in *%o0* and above. Furthermore, the application libraries expect that the carry condition code bit indicates success (cleared) or failure (set) of the system call. If the call failed, the return value is assumed to be the corresponding negative *errno* value.

The system trap handler utilizes the generic trap-entry and exit routines described above to save and restore user state. The addresses of all system call routines are stored in a table in *syscall/syscall.s*. After saving the current process state, the handler multiplies the system call number by four, and checks if the resulting table index is outside the table boundaries. If this is the case, it calls the *sys_illegal* routine which prints a message and sends a *SIGSYS* signal to the current process. This routines address can also be used in the system call table for unused entries. If the system call number is valid, the routine moves the system call arguments from the input to the output registers (necessary because the trap entry code allocated a new register window) and jumps to the address specified in the system call table.

Upon return from the system call, the handler checks if the result is positive (success) or negative. If the system call failed, it reads the old processor status word from the stack, clears the carry-bit of the saved condition code register, writes back the modified processor status word and inverts the return value (which will become *errno*) When returning to the user process, the library will check the carry bit, copy the return value into the application variable *errno* and return -1 to the user program.

If the system call succeeded, the handler routine clears the carry bit by reading *TState[TL]*, clearing the appropriate bit and writing the modified processor status word back. In either case, since the trap-exit code restores the input registers (which will become the output registers of the user process), the system call handler needs to store the return values (*%o0, %o1*) on the stack from where the trap-exit code will restore them.

The system call jump table is Solaris compatible, which means the system call numbers and arguments/return values are believed to be the same as in the Solaris operating system. As a result, any Solaris-compliant static library can be used to link ML-RSIM applications.

4.7 Interrupt Handler

files: interrupts/traps.s, interrupts/machine/intr.c
interrupts/trap.h, machine/intr.h

Most external interrupts (except for the clock interrupt) are handled by a generic chained interrupt mechanism. This mechanism allows devices and device drivers to share interrupt vectors. All interrupt handlers for a particular vector are chained through a linked list. A list entry consists of a pointer to the handler function, an argument pointer which is set when the interrupt is established, a counter and a pointer to the next element. The root pointers for these linked lists are maintained in an array which holds one entry for each external interrupt.

Device drivers can establish interrupt handler routines by calling *intr_establish* with the interrupt number, handler function and an argument pointer as arguments. This routine allocates a new linked list element, copies the arguments into the element and appends it to the appropriate linked list. The routine returns a pointer to the newly allocated list entry.

Similarly, interrupt handlers can be removed from the linked lists by calling *intr_disestablish*. This routine takes a pointer to the list element to be removed (the return value of *intr_establish*). It searches through the appropriate linked list until the element is found and removes it.

Each interrupt vector may alternatively be used as a fast interrupt without the ability to chain various interrupt handlers together. A fast interrupt handler jumps immediately to the interrupt routine instead of traversing a linked list of interrupt handlers. The routine *intr_establish_fast* establishes such a fast interrupt handler by overwriting the address of the interrupt handler routine in the low-level trap table, using the kernel patch facility. Fast interrupt handlers can not be disestablished. The routine *intr_handler* contains the actual interrupt handler code. It takes as arguments the trap type and a pointer to the saved processor registers. It walks through the linked list corresponding to the interrupt number, calling each interrupt handler in turn, passing the interrupt number, register pointer and argument pointer to it.

The simulator triggers interrupt 1 (power failure) when the user has interrupted the simulation by pressing Ctrl-C. The interrupt handler prints out a message acknowledging the event, sends a *SIGINT* signal to all processes in the system and sets the *power_fail* flag. The second time the power failure interrupt is raised, the trap handler sends a SIGKILL to all remaining processes to force termination. This causes all processes to exit, at which point the *init* process shuts down the system by unmounting the file systems and synchronizing the disks. The *power_fail* flag is checked during initialization (before any user processes are created) and causes the initialization routine to skip any further initialization and shut down immediately.

5 System Calls

This section describes miscellaneous system calls. System calls that belong to a larger subsystem, like the shared-memory calls, or file-I/O calls, are described in later sections.

5.1 Fork System Call

files: mm/syscalls.s
mm/mm.h

The system call routine first allocates a new process structure by calling *create_process*. The system call routine then copies the parents page table mappings (*copy_pagetable*), and the parents memory (*copy_memory*), process attributes, signal dispositions and open file descriptors. The new process is then made a child of the current process by inserting it into the current processes list of children.

The new process returns the parent process ID in register *%o0* and the value 1 in register *%o1*. This is accomplished by modifying the saved input registers in the process structures. Also, the stack pointer, return address and current window pointer are initialized to the same values as in the current process. When a context switch to the new process occurs, these values will be restored and the process continues execution at the same point as the parent process. After acquiring the necessary locks, it inserts the new process in the process list and in the run-queue.

The system call returns the child's process ID in register *%o0* and the value 0 in register *%o1* to the parent process (the original caller).

5.2 Indirect System Call

files: syscall/syscall.s

For backward compatibility reasons, several system calls use call number 0 (*%g1* = 0), in which case the actual call number is passed in *%o0*.

System calls of this kind are dispatched to the same exception handler, which looks up the address of the indirect call routine in its call table, just as described for normal system calls. It then jumps to the indirect call handler (entry 0), which first moves the system call number from *%o0* to *%g1*, moves all arguments in *%o1*-*%oN* to *%o0*-*%oN*-1 and jumps to the normal system call handler.

5.3 Illegal System Call

files: syscall/syscall.s, interrupts/traphandlers.c

This routine is the default for all unused system call table entries. In addition, it is called when the system call entry code detects that the system call number is outside the bounds of the system call table. The routine prints a diagnostic message and sends the *SIGSYS* signal to the current process.

5.4 Brk System Call

files: mm/syscalls.c
mm/mm.h

This system routine is called whenever the heap must be extended towards higher addresses, due to calls to *malloc*, or can be shrunk due to calls to *free*. It takes the desired break value as argument and returns 0 upon success, or -*errno* upon failure.

If the requested increase is greater than the current break value (the upper bound of the data segment), the routine allocates the required number of physical pages and installs the corresponding mappings. Otherwise, it uninstalls the required number of page table mappings, but without actually deallocating physical memory in the simulator.

5.5 Process Attribute System Calls

files: syscalls/sysinfo.c
kernel/process.h

Each process has several attributes associated with it which are used for various permission checks. These attributes include the process-ID, process group ID, session ID, user ID, user group ID and a list of user groups. Generally, these attributes are inherited during *fork*, but system calls exist to manipulate them. The process group leader is the process whose process ID is equal to the process group ID. Similarly, the session leaders process ID is equal to the session ID.

The system calls *getuid* and *getgid* return the user and user group ID of the calling process. The complementary system calls *setuid* and *setgid* are not implemented since they can only be executed by the super-user or under certain conditions by an ordinary user process, and they are not expected to be used in most programs.

The system call *getpid* returns the callers process ID in register *%o0*, and the parents process ID in register *%o1*. The C-library uses this system call to implement the *getpid* and *getppid* routines.

Most of the attribute manipulation functionality is combined in the *pgrpsys* system call. The first argument specifies the operation to be performed. If the command is *GETPGRP* (0), the system call returns the callers process group ID. The command *SETPGRP* (1) makes the caller a process group and session leader by changing its group and session ID to its process ID, unless the caller is already a session leader (pid == sid). The system call returns the new group ID. If the command is *GETSID* (2), the call returns the session ID of the process whose ID is equal to the second argument, or the session ID of the calling process if the argument is 0. The command *SETSID* (3) makes the calling process a session leader and group leader by setting its session and group ID to its process ID, unless the caller is already a group leader. If the command is *GETPGID* (4), the system call returns the group ID of the process whose ID matches the second argument, or the group ID of the calling process if the argument is 0. The command *SETPGID* (5) sets the group ID of the process whose ID matches the second argument to the group-ID specified in the third argument. If the process ID is 0, it sets the group ID of the calling process. If the group ID is 0, it makes the process a group leader by setting its group ID to its process ID.

The list of user groups a process belongs to can be retrieved by the *getgroups* system call, and manipulated by *setgroups*. These calls take the number of elements and a list of user group IDs as arguments. Only the superuser (UID is 0) is allowed to modify group membership.

The system call *times* returns the CPU time used by the current process in user mode and kernel mode as well as the total CPU times used by all its children. The routine simply copies the relevant fields from the process structure into the structure provided by the application, and returns the current value of the *tick* counter.

5.6 System Attribute System Calls

files: syscall/sysinfo.c

Several system calls can be used to retrieve general system information, system limits and configuration parameters. The system call *sysconfig* provides general system parameters, it is used by the C library routine *sysconf*. It takes an integer as argument, which indicates which parameter to return. Most of the configuration parameters are fixed in this version, such as the number of groups per user, the number of open files.

The *uname* system call returns general information about the system, such as host name, architecture, OS and OS version. This information is returned in a structure that is passed as a pointer to the system call. The obsolete system call *utssys* serves a similar purpose, depending on its third argument it returns *uname* information or other, as of yet unknown values.

Sysinfo is the replacement of the *uname* system call. It returns similar information, such as host name, system name, OS version, hardware serial number and the RPC domain name. It takes as arguments a command identifier, a pointer to a buffer and the buffer size, and returns the ASCII representation of the requested parameter in the buffer.

Per-process resource limits can be retrieved with the *getrlimit* system call, and modified with the *setrlimit* system call. Both calls take a resource identifier and a *rlimit* structure as arguments. The *rlimit* structure contains a hard limit and soft limit value. Processes may reduce the hard limit and may change the soft limit to any value less then or equal to the hard limit. Currently, most resources are either unlimited or have a fixed limit that can not be changed.

5.7 User-Context System Calls

files: kernel/context.c, kernel/context_asm.s

The system calls *get_context* and *set_context* allow applications to implement some form of application-controlled context switching. The *ucontext_t* structure contains fields that describe the signal mask, user-stack and CPU state. Flags indicate which of these fields are valid.

The system call *get_context* stores the current context in the structure that is provided as argument. It reads the current signal mask from the process structure and puts it in the first word of the *ucontext* signal-mask field. Note that Solaris supports up to 128 signals, whereas the simulator OS supports only 32 signals (as did original Unix versions), hence the remaining three mask words are set to 0. The stack field is filled with the base address of the user-stack (*stack_low* in the process structure) and with the length of the user-stack.

The CPU context is divided into general-purpose (integer), register window, floating point and extra context. The general-purpose field contains space for the global and output registers as well as for the PC, next-PC, processor status word and Y-register. These fields are written with the values that have been stored on the kernel stack upon system call entry.

The register-window descriptor is currently not supported, it is set to NULL. This indicates that the register windows have been saved in the usual place (or will be saved through the window trap mechanism). If it is not NULL, this field points to a structure that describes where the register windows have been saved.

The floating point context contains fields for the floating point data registers and the status register, as well as some elements that describe the state of a floating point queue. This queue appears to be not implemented in any SPARC processors so far, and the corresponding fields are not used. The floating point data and status register fields are written by an assembler routine.

The *ucontext* structure provides for some implementation dependent state to be saved in the extra-elements. If the extra-ID is equal to a magic number, the pointer element points to a data structure that describes additional CPU state. This is currently not supported.

The routine *set_context* restores the context provided as argument, effectively performing a context switch. It writes the signal mask into the process descriptor mask field, writes the integer registers into the corresponding fields on the kernel stack where they will be restored into the CPU registers upon system call return, and restores the floating point unit state. Note that the *ucontext* structure contains a set of flags that indicates which of the context-components are valid, the system call restores only the valid portions. Since the *setcontext* routine is also used in the process of delivering signals to processes, it must not have a return value of its own, as this would overwrite the return value of a system call that completed before the signal was delivered. For this reason, the routine *do_setcontext* returns the contents of registers *%o0* and *%o1* as they appear in the *ucontext* structure.

Note that the PC value of a context always points to the instruction where execution will resume when the context is restored, in other words it points to the instruction following the trap instruction that causes the system call. The *sys_getcontext* routine stores the next-PC as current PC in the context structure, for this reason. When the context is restored, the *sys_setcontext* routine restores the PC as the next-PC, since it returns to user code via the *Done* instruction. If, however, the context is saved as part of the signal delivery procedure, execution is eventually to resume either at the current instruction or at the following instruction, depending on whether the signal is delivered following a system call or an interrupt. The signal delivery code communicates these cases through a flag to the *do_signal* routine, which forwards it to the *sys_getcontext* routine. If the signal is delivered following an interrupt, the saved context must store the current PC and not the next-PC in order to retry the interrupted instruction when the signal handler returns.

5.7.1 Time System Calls

files: syscall/misc.s
kernel/kernel.h

The *time* system call simply reads the current time from the kernel time variable and returns it in register *%o0*.

The high-resolution clock system call returns time information at the highest possible resolution. The meaning of the time value depends on the clock-ID specified as argument. Currently, only clock-IDs 0 and 3 (real-time clock) are supported. For these clocks, the *clock_gettime* system call utilizes the *gettimeofday* system call.

The *clock_gettime* system call returns the number of seconds (in *%o0*) and microseconds (in *%o1*) since 1970. The first value can be obtained from the kernels time variable, similarly to the time() system call. The processors cycle counter is used to obtain the high-resolution portion of the return value. The cycle counter starts at 0 when the processor is initialized, and is reset to 0 once a second when the kernel time variable is incremented. The microsecond value can be computed as current cycle count modulo clock frequency (in MHz).

The *clock_getres* system call returns the resolution of the specified clock. It can be computed as 1e9 divided by the clock frequency. Note that this system call stores the return value in the structure that is passed as an argument, unlike the *clock_gettime* call which returns the result in registers.

6 Signals

Each process contains several variables that define how signals are handled. The signal-variable is a bit vector of pending signals. This includes signals that have been send but not yet been delivered to the process, and signals that are currently blocked. The signal mask variable is a bit-vector of signals that the process does not want to handle at this point, they are delivered as soon as they are unblocked. Note that both bit vectors provide for 32 signals, while Solaris allows up to 128 signals.

The array *signal_action[]* stores the signal disposition for each signal, along with flags controlling signal delivery and a signal mask that is in effect when the signal is delivered. Note that both signal bit vectors and the *signal_action* array are indexed starting at 0 (the normal C convention), while the first signal number is one. Thus, the system subtracts one from the signal number before using it as an index.

In addition, each process structure contains an array of signal information structures, one for each signal. When a signal is posted, the signal information is copied into the array entry corresponding to the signal, from where it is copied to the user stack when the signal is delivered.

6.1 Initialization

files: *kernel/signals.c*
kernel/signals.h

The routine *init_signals* resets the signal-related variables of a process to their default values. It clears the bit vectors of pending and blocked signals, sets the signal dispositions to *SIG_DFL* and clears the flags and mask fields of the *signal_action* array. This routine should only be called when the first process (*init*) is created, or before a process executes exec. Normally, processes inherit the signal settings from the parent process. The routine *copy_signals* performs this copy operation. In addition, the *kernel_init* routine copies the signal trampoline code onto the top of the user stack, from where it will be inherited by all its children.

6.2 Signal System Calls

files: *kernel/signals.c*
kernel/signals.h

The *sigaction* system call provides the general interface to control signal handling. It is used by the C-library to install signal handlers. The call takes as arguments a signal number and two pointers to *sigaction* structures. If the first pointer is not NULL, it sets the signal handler, flags and mask to the values provided in this structure. Before updating any signal handling variables, the system must acquire the *signalled*. In addition, if the new signal disposition is *SIG_IGN*, the system clears this signal in the signal bit vector (unless the signal is *SIGCHLD*). If the second pointer is not NULL, the system call stores the old signal handler, flags and mask in that structure.

The system call *sigprocmask* is used to inquire about the current signal mask, and/or to install a new signal mask. It takes as arguments a flag indicating how the new mask is to be applied, and two pointer to signal mask structures. Note that these structures provide space for 128 signals (4 words), while the system uses only the first word. If the second pointer is not NULL, the call returns the current signal mask in that structure. If the first pointer is not NULL, the system call either adds the new mask to the existing signal mask (how = *SIG_BLOCK*), removes the signals specified in the new mask from the existing mask (how = *SIG_UNBLOCK*) or replaces the existing mask with the new mask (how = *SIG_SETMASK*). In any case, the system call silently enforces that *SIGSTOP* and *SIGKILL* can not be blocked.

The system call *sigpending* performs two functions, depending on the first argument. If the argument is *SIGPENDING* (0), it returns the bit vector of pending signals (sent but blocked) in the structure pointed to by the second argument. If the first argument is *SIGFILLSET* (1), it returns a mask of valid signals.

The *signal* system call provides an alternative, but less flexible interface for signal control. It takes as arguments a signal number, which may include various flags in the upper bits, and a handler address. It sets the signal disposition for the specified signal to the handler. By default, the signal handling flags are set to *NODEFER* and *RESETHAND*, which means that the signal disposition is reset to *SIG_DFL* when the signal is delivered, and that the signal is not blocked while the handler is executing. The optional flags provided in the upper bits of the signal number can specify that the signal should be blocked when the handler is executing (*SIGDEFER*). Other flags specify that the signal should be added or removed from the signal mask, but no handler is installed, or that the signal is to be ignored. Finally, the flag *SIGPAUSE* causes the signal to be removed from the signal mask and the process to suspend execution until a signal arrives. This is currently not implemented.

Normally, when *signal* is called, the C library passes its own signal handler wrapper to the system call *sigaction*. This wrapper routine provides the trampoline code necessary to resume after the signal handler has executed. If, however, applications trigger the *signal* system call directly, the system must provide this trampoline code. To distinguish these two cases, signal handlers installed by the *signal* system call are marked with a special flag *SA_TRAMPOLINE*.

The system call *sigaltstack* allows applications to specify an alternate signal handling stack, and/or to get the current signal stack settings. A stack is specified by its start address and size, where start address is the lower boundary of the stack. Both values are recorded in the process structure. A flag specifies whether the alternate stack is enabled or disabled. In addition, the *sigaction* call can enable or disable the alternate stack on a per-signal basis.

The system call *sigsuspend* suspends the current process until a signal is received. Records the process signal mask upon call entry in the process structure, replaces it with the mask passed in as an argument and puts the process to sleep, waiting on its own process structure. When it is woken up, it checks if a signal has arrived that is not blocked, in which case it returns with the *EINTR* error code. If the process had been woken up for reasons other than a signal, or the signal is blocked, the routine continues sleeping. The signal handling flag in the process structure is set to *SA_OLDMASK* to instruct the signal delivery code to restore the process signal mask before to the original value.

6.3 Signal Transmission

files: kernel/signals.c
kernel/signals.h, kernel/process.h

Signals are delivered either by the kernel in response to an event, or by a process through the *kill* system call. This call takes as arguments the process ID and signal number. If the process ID is greater 0, the system sends the signal to the process whose ID corresponds to the argument. If the process ID is 0, the signal is sent to all processes in the same group as the calling process. The system call searches sequentially through the process list and checks the group IDs. If the process ID argument is negative, but not -1, the signal is send to all processes in the group whose ID is equal to the absolute value of the process ID argument. Finally, if the process ID specified is -1, the signal is send to all process with a user-ID equal to the calling processes user ID. Note that it is not possible to send signals to the *init* process.

For every process that is to receive a signal, the system calls the *send_signal* routine. If the signal causes the process to continue execution (*SIGKILL* or *SIGCONT*), it wakes up the process and removes signals that would cause it to stop (*SIGSTOP*, *SIGTSP*, *SIGTTIN*, *SIGTTOU*) from the signal bit vector. If, on the other hand, the signal causes the process to stop (see above list), it removes the signal *SIGCONT* from the list of pending signals. Finally, the routine checks if the signal would be ignored by the process and posts the signal only if this is not the case. If the process was sleeping in an interruptible state, it is awoken. In addition, the *send_signal* routine copies the signal information provided as argument into the corresponding *signal_info* entry in the process structure.

6.4 Signal Delivery

files: kernel/signals.c, kernel/context.c, kernel/context_asm.s
kernel/signals.h, kernel/process.h

When returning from a system call or an interrupt/trap to user mode, the system checks for any pending signals. If the signal bit vector is not zero, it calls the *do_signal* routine which performs most of the signal delivery.

This routine takes as arguments a pointer to the process structure, the signal bit vector and a pointer to the user state saved on the kernel stack. It first determines the signal number from the signal bit vector by checking each bit that is not blocked starting at bit 0. If all pending signals are masked, the routine returns immediately.

The routine then prepares for the signal delivery by allocating space on the user stack to hold the signal handler arguments. The signal handler takes the signal number, a pointer to a *siginfo* structure and a pointer to a *ucontext* structure as arguments. If there is insufficient space on the user stack for these data structures, the system grows the stack as necessary, or aborts the process if the stack exceeds the limit. The routine then copies *siginfo* structure and the current user context into the *ucontext* structure (by calling *do_getcontext*) on the user stack. An additional parameter indicates if *do_signal* is called from a trap/interrupt or a system call. In case of an interrupt, the signal handler trampoline code should return to user mode with the *RETRY* instruction. Since the user context is restored by a system call, which returns with a *Done* instruction, the application would skip an instruction. For this reason, the trap *PC* and *nPC* are set to the previous instruction after the current user context is saved.

Next, the routine clears the signal and installs the signal mask specified in the *signal_action* array for this signal. Additionally, if the *NODEFER* flags was not set, it adds the current signal to the mask. Furthermore, if the *RESETHAND* flag was set, it resets the signal disposition to *SIG_DFL*.

If the signal disposition is set to *SIG_DFL*, and the default action for the signal is *kill*, the process is aborted by calling *do_exit*. If the signal is *SIGCHILD* and the process ignores the signal (disposition is *SIG_IGN*, or *SIG_DFL* and default action is *ignore*), the child is removed from the system on the parents behalf. Finally, if the process ignores the signal, the routine returns.

The actual signal delivery is performed by modifying the user state saved on the kernel stack. The routine sets the argument registers so that they contain the signal number, a NULL pointer (siginfo*), and the pointer to the *ucontext* structure on the user stack. It sets the *PC* and *nPC* to point to the signal handler, and adjusts the stack pointer. Upon return from the system call, the trap-exit code will restore the user state with these values and jump to the signal handler. The C library wraps user signal handlers in a routine that calls *setcontext* with the user context as argument, which causes the system to restore the user state and continue normal execution. In order to support applications that install signal handlers directly, the system marks such signal handlers with a special flag. In this case, the PC points to the trampoline code located at the top of the user stack. The fourth signal handler argument is the address of the actual signal handler, which is used by the trampoline code to execute the signal handler.

7 Timers

files: devices/realtime_clock.c, kernel/itimer.c, kernel/timer.c
kernel/process.h, kernel/itimer.h, kernel/timer.h

The system provides three different timer facilities, all of which are controlled by two system calls. The realtime timer is incremented in real time, regardless of which process is executing. The process receives a *SIGALRM* signal upon expiration of this timer. The virtual timer is incremented only when the process is executing in user mode and a *SIGVTALRM* signal is send upon expiration, while the profile timer is incremented when the process is executing in user or kernel mode, it sends *SIGPROF* when it expires.

Each process structure contains several fields to manage these timers. The *value* field is loaded with the current timer value, it specifies how many clock ticks after being loaded the timer expires. The *incr* field specifies a timer interval, it is used to reload the timer when it expires. This field may be 0, in this case the timer expires only once. Finally, the *current* field is used to keep track of the actual timer, it is decremented by the clock-tick handler routine. When this field reaches 0, the corresponding signal is send to the process and the timer is reloaded with the interval value.

Since the realtime timer is independent of the process state, it can not be managed on a per-process basis. Instead, the system maintains a global list of timers which is ordered according to expiration time. Each timer is described by a structure that contains the expiration time (in absolute clock ticks), a pointer to a function to be called when the timer expires and a 32 bit argument for this function. These structures are managed in a pool of free timer elements. Functions exist to remove and insert timer descriptors from the free list as well as to add and delete timers from the timer queue. If the free list is empty and another timer is requested, the allocation routine allocates a new timer structure in kernel memory.

7.1 Interval Timers

files: kernel/itimer.c
kernel/itimer.h

The system call *setitimer* can be used to start and stop timers. It takes as argument a structure that contains the timer value and interval as *timeval* structures (consisting of seconds and microseconds). The specified values are converted to clock ticks, rounded up to the next higher clock tick and stored in the process structure fields corresponding to the desired timer. Optionally, the system call stores the old values in another structure that can be passed as second argument. Note that when setting a timer it is also necessary to load the current timer value with the new value in order to avoid spurious timer expirations from old values.

For the realtime timer, the system call also allocates a new timer descriptor if the process did not have one, sets up the descriptor (existing or newly allocated) with the expiration time, the pointer to the *do_itimer_real* function and the process pointer and adds it to the global timer list. If the timer value passed to the system call is 0 (indicating that the timer is to be deactivated) and the process has an active realtime timer, it returns the timer to the free-list.

The *alarm* system call is implemented as a wrapper routine that initializes a *itimer* descriptor with the timeout value specified as parameter and then calls *sys_setitimer*. This call will only be used by applications that use the system call directly, as the C-library translates the *alarm* library call to a call to *setitimer*. The system call *getitimer* simply returns the current timer values in a structure that is passed as argument.

7.2 Realtime Timer Event Handler

files: kernel/itimer.c

The clock interrupt handler routine calls the timer handler function for every expired timer, and also deletes the timer from the global list before calling the function. In case of the realtime timer, the function *do_itimer_real* first sends the signal *SIGALRM* to the process (the process pointer is passed as an argument to this routine). If the timer interval value is not zero, it loads the timer descriptor with a new expiration time and reinserts it into the list, otherwise it returns the descriptor to the free-list and sets the timer value in the process structure to 0, indicating that the timer is not active.

8 Shared Memory

files: syscall/shmsys.c
syscall/shmsys.h, mm/mm.h

The shared memory interface is with a few exceptions System V compliant. Shared memory segments are allocated by calling *shmget*. A new segment is created if either the key is equal to *IPC_PRIVATE*, or (*flag & IPC_CREAT*) is true and a segment with that key does not exist. If a segment with the given key already exist and (*flag & IPC_EXCL*) is true, the system call fails, otherwise it returns the ID of the existing segment.

shmat maps an existing shared memory segment into the calling process' address space. If the address is NULL, the segment will be mapped at an arbitrary address. Mapping a segment to a specific address (as defined in System V) is not supported.

A shared memory segment can be unmapped from a process' address spaces by calling *shmdt*. The *shmctl* system call can be used to get or set the status of a shared memory segment or to remove a segment.

8.1 Segment Descriptors

files: syscall/shmsys.c
syscall/shmsys.h, mm/mm.h

The shared memory descriptor table is a dynamic array of segment descriptors, indexed by the shared memory segment ID. Each descriptor contains permission information such as user and group ID of the creator and protection bits, the segment key and size, process IDs for the creator and the last access and times of creation, last attach and last detach operation. These fields correspond to the elements of the *shmid_ds* structure used by the *shmctl* system call. In addition, each segment descriptor also stores a pointer to the corresponding segment. The descriptor array contains initially 16 elements, but grows if more segments are requested by processes.

Shared memory segments are allocated sequentially, without consideration for fragmentation due to segments that have been removed. The system call *shmat* simply allocates pages starting at *shared_high*. Furthermore, *shmdt* only removes the page table mappings but does not free the memory. However, if a segment has been detached by all processes but not yet been removed, and is attached again, the already allocated pages will be used.

8.2 System Calls

files: syscall/shmsys.c
syscall/shmsys.h

Shared memory segments are created with the *shmget* system call. If the specified key is *PRIVATE*, the routine searches for an unused entry in the array of segment descriptors. If no unused segment is found, it grows the descriptor array to twice its original size and returns the first new segment.

If the *create* flag was specified, the system call first checks if a segment with the specified key already exists. If one was found, the process has not specified the *excl* flag and the segment is large enough, it is returned. If no segment was found, the system call allocates a new segment descriptor, while possibly growing the descriptor array.

If a new segment was created, either because the *private* key or a unique key was specified, the new segment descriptor is initialized with the callers user and group ID, the access mode is set to the value passed to the system call and the creation time and process ID are set appropriately. Finally, if neither the *private* key nor the *create* flag where passed to the system call, it simply searches for an existing entry that matches the key, and returns it if found.

The *shmat* system call currently supports only the attachment of memory at an address determined by the system. Before attaching the shared memory segment, the system call checks if specified segment is valid, if the system has sufficient shared memory pages available and if the calling process is allowed to attach the segment. When all checks are passed, the routine increments the attach-count in the segment descriptor, sets the last-attach time value to the current time and saves the calling process's ID in the descriptor. If the process is the first to attach to the segment (attach-count is 1), it allocates new shared memory pages by incrementing the *shmem_high* pointer, saves the base address in the descriptor, and maps the pages into the user process, otherwise it just maps the pages.

Since the only argument to the *shmdt* routine is the base address of the shared memory segment, the system call first scans the list of segments to determine which one to detach. It then saves the callers process ID and current time in the segment descriptor, decrements the attach-count and unmaps the segment from the current address space.

The *shmctl* system call performs several different operations, depending on the second argument. If the command is *IPC_STAT*, it copies the relevant fields from the segment descriptor into the user-provided *shmid_ds* structure. The *IPC_SET* command sets the UID, GID and access-mode fields of the descriptor with the values provided in this structure. Segments are removed with the command *IPC_RMID* by setting the key, mode and size fields to 0. The lock and unlock commands are currently not supported.

9 User Statistics

files: *syscall/userstat.c, syscall/userstat_asm.s*
syscall/userstat.h

The user statistics feature allows any software running on the simulator to create and control statistics gathering objects inside the simulator with very little overhead. Statistics objects have unique names that are provided during allocation and are identified by a non-negative number. Note that this is a simulator-only feature, it does not correspond to any real hardware and is not available on native Solaris systems.

Statistics objects are allocated in the routine *sys_userstat_get*. This routine is both a general kernel routine and a system call, since application programs should also be able to create statistics objects. The routine takes the statistics type and a name string as arguments. It copies the provided name string into a page aligned buffer and translates the virtual buffer address into a physical address, which is then passed to the simulator trap. The routine returns a non-negative integer upon success, which is used for subsequent *sample* calls to identify the statistics object. In case of a failure, it returns the negative *errno* code.

Statistics objects receive samples through the *userstat_sample* simulator trap. The trap takes the object ID (returned when allocating the object) and a type-specific value as arguments. The semantics of the trap and the user-provided value depend on the particular statistics type.

The kernel may call the *sys_userstat_get* routine directly, whereas user programs must use the system call stub *userstat_get*. This routine and the *sample* trap stub are compiled and linked into a application library.

10 File Systems

The filesystem is almost completely based on NetBSD code. Modifications where only made as a result of the differing process structures and memory management. The following section describes only briefly the virtual filesystem layer of BSD, since this is discussed in more detail in other publications.

10.1 Virtual Filesystem Layer

The virtual filesystem is an object-oriented layer that isolates the system call code from the particular filesystem implementations. The basic data structures of the virtual filesystem layer are the vnode, lists of supported file systems with pointers to filesystem management routines and a global list of vnode operations supported by each filesystem.

10.1.1 VNodes

files: sys/vnode.h, sys/vnode_if.h

A vnode is a structure that describes a file that is currently in use by the system. This includes regular files, directories, devices or sockets. Each vnode contains a pointer to the *mount* structure that describes the filesystem that contains the object, a pointer to a set of vnode operations (read, write etc.), the object type, and clean and dirty buffer pointers. Finally, each vnode contains a pointer to filesystem-private data which may for instance contain the inode number.

Vnodes are managed as a fixed-sized pool, and are recycled when they are no longer needed. Vnodes that describe regular files or directories are also part of a linked list in the filesystem that contains the object. This linked list is used to flush all buffers and properly close the objects when the filesystem is unmounted.

10.1.2 File Descriptors and Current Directories

files: fs/file.c, fs/filedesc.c, kern_descrip.c
sys/file.h, sys/filedesc.h

The kernel maintains a list of open files for each process in the process structure. In Lamix, the list size is fixed to 64 entries. Each file descriptor is a structure that contains various file mode flags, list pointers, the current file offset, a pointer to the process credentials, a pointer to the file operations and a pointer to the associated vnode or socket structure.

When a new file is opened, the kernel uses the file descriptor with the lowest available number. Upon fork, the file descriptor list is copied to the child process, which means that the child process inherits all open files including file offset and access attributes.

In addition to the file descriptor list, each process maintains a current working directory and a root directory. Normally, the root directory is the root vnode of the system, but the *chroot* system call can be used to define a different directory as the root for the process. Both root directory and current directory are represented by a pointer to the respective vnode.

10.1.3 Name Cache and Name Lookup

files: fs/vfs_lookup.c, fs/vfs_cache.c
fs/namei.h

One of the central responsibilities of the virtual filesystem is the translation of path names into vnodes with the associated vnode allocation and locking. The *namei* routine performs this operation with a number of variations. Essentially, the routine calls the *lookup* routine which translates the path name into a vnode. If the path name does not start with at the root directory, it determines the current directory vnode, otherwise the global root vnode is used as starting point. If the vnode returned by the lookup routine is a symbolic link, the filesystem-specific *readlink* routine is called which retrieves the path name of the object that the link points to, and another lookup is performed. For each iteration, the routine increments a loop counter in order to detect circular links.

The *lookup* routine translates a path name step by step into a vnode. For every path name component, it calls the filesystem specific lookup routine (based on the current directory vnode). If the vnode returned by the lookup is a mount point, it finds the root vnode of the file system that is mounted at this directory. Flags determine if the vnodes are to remain locked after the translation and if an object is to be created if it is not found (needed for create system call).

To speed up the path name translation process, the virtual filesystem also maintains a name cache. It caches translations from path names to vnodes. In addition, it supports negative caching, in which case an entry indicates that the object does not exist. Entries are allocated by calling *cache_enter* with the path name and vnode as arguments. This routine is usually called by the filesystem specific lookup routines after a successful lookup, or when the object was not found (negative caching). Similarly, before reading directory entries from disk, the lookup routines access the cache to check if a vnode already exists for a path name component (routine *cache_lookup*).

10.2 Buffer Management

files: kernel/kern_allocsys.c, fs/vfs_bio.c, machine/machdep.c
sys/buf.h

Buffers are used to transfer data between block I/O devices and user space, and act as a cache of recently used blocks. Unlike other Unix variants, the BSD filesystem uses a fixed size buffer pool. The number of buffers as well as the number of physical pages is determined at boot time. The amount of physical memory used for the buffer cache can be defined at compile time through the configuration file. If no size is specified, the routine *allocsys* assigns 10% of the first 2 MByte of physical memory plus 5% of the remaining memory for the buffer cache. Similarly, the number of buffers can be set at compile time. If not set, the number of buffers is equal to the number of buffer cache pages, which means that every buffer is initially assigned exactly one page. The kernel also recognizes command-line parameters that specify the number of buffers, number of buffer pages or size of the buffer cache in percent main memory.

The following figure shows the virtual address layout of the buffer cache as it is created by the *cpu_startup* routine.

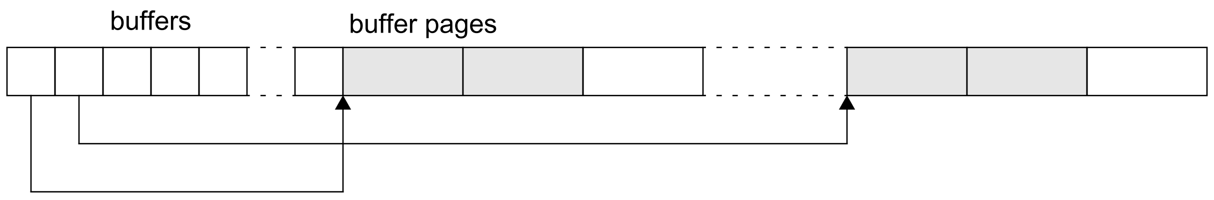

Figure 21: Buffer Cache Layout

Each buffer is assigned a virtual address region of size *MAXBSIZE*, but this region is not fully populated with physical memory. Initially, the buffer pages are equally distributed among the buffers. If a buffer needs more space than it currently has, it ‘steals’ pages from other buffers by simply remapping the physical memory.

Buffers are requested by a filesystem through the *getblk* routine. This routine checks if a buffer for the given vnode and block number already exists in the buffer cache. If no buffer is found, a new buffer is allocated by calling *getnewbuf*. Finally, the routine *allocbuf* is called to insure that the buffer has the required size. An empty, clean buffer of a given size can be requested by calling *geteblk*. This routine is similar to *getblk*, except that it does not check the buffer cache if a buffer for that vnode already exists.

Allocating a new buffer may involve writing back dirty buffers. The routine *getnewbuf* first checks if a buffer is available from the free-list. If this is not the case, the LRU and age-list are checked. The LRU list contains all used buffers in the order in which they have been used, whereas the age list contains buffer in the order in which they have been requested initially.

The external interface to the buffer cache is through the *bread* and *bwrite* routines. *bread* first requests a buffer for the vnode with a specified block offset and size. The buffer may be found in the buffer cache, or a new buffer may be allocated. The routine then checks if the buffer has the required amount of data, and if not it initiates an I/O transfer to fill the buffer. The routine *bwrite* operates similarly, except that it marks the buffer as dirty.

10.3 Filesystem Types

Currently, Lamix supports three different file systems: the original BSD filesystem FFS (also known as UFS), the log-structured filesystem LFS and a new HostFS file system that imports the simulation host filesystem into the simulated kernel.

Each filesystem provides a list of supported filesystem operations (such as mount/unmount, sync etc.) and a list of file operations. During kernel configuration, a list of supported file systems is created. When the kernel boots, the routine *kernel_vfs_init* scans this list and attaches each file system, which includes incorporating the file system operations into a global list of routines.

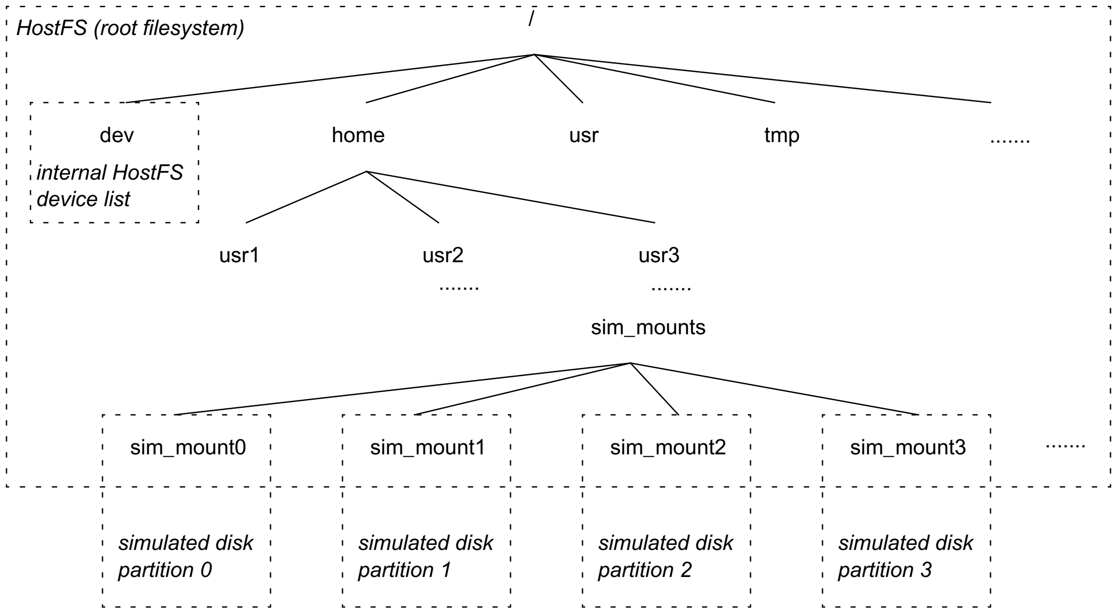
HostFS (root filesystem)
/
dev
internal HostFS device list
home
usr
tmp
....
usr1
usr2
usr3
....
....
sim_mounts
sim_mount0
sim_mount1
sim_mount2
sim_mount3
....
simulated disk partition 0
simulated disk partition 1
simulated disk partition 2
simulated disk partition 3

Figure 22: File System Tree

Normally, the host file system is mounted as the root file system, with the hosts root being also the Lamix root. Other file systems, which reside on simulated disks, are mounted by the routine *kernel_mount_partitions*. For each disk device that was detected during autoconfiguration (including software raid devices and concatenated disks), the routine opens the device and reads the partition table. Every partition that contains a known file system (FFS or LFS) is mounted at the *sim_mounts/sim_mountXX* mount points in the ML-RSIM tree. For instance, the first partition would be mounted on */home/<user>/ml-rsim/sim_mounts/sim_mount0*, the second partition in turn is mounted on */home/<user>/ml-rsim/sim_mounts/sim_mount1* and so forth. The routine continues until either all partitions of all disks are mounted or until a mount attempt fails. Note that a failed mount does not abort the simulation.

10.4 Host File System

The host file system is a local file system that gives simulated applications access to the file system of the simulation host. The file system does not use buffers and does not do any real I/O, instead data is transferred instantaneously between the application buffer and the host file system through the file-I/O simulator traps. Normally, the host file system is mounted as root file system to reflect the host directory structure accurately within the simulated system.

10.4.1 Inode Management

files: ufs/hostfs/hostfs_inode.c,, ufs/hostfs/hostfs_ihash.c, ufs/hostfs/hostfs_lookup.c
ufs/hostfs/hostfs_vnops.c
ufs/hostfs/hostfs.h, ufs/hostfs/inode.h

The host file system is different from other file system in two major aspects. First, it does not reside on a single device but on a collection of devices, since it comprises the entire directory structure of the simulation host, including all mounted file systems. Second, it does not use the buffer cache but transfers data instantaneously to and from user space.

Similar to the UFS file system, files and directories are represented by inodes. Inodes are uniquely identified by the host device and inode number. This identification is used when inodes are looked up in the open-files hash table. In addition to this identification, inodes contain various flags, the node and device number of the parent directory, a pointer to the vnode associated with the object and a pointer to a file lock structure.

Most importantly, *HostFS* inodes contain a file handle that is used for physical access to the file through the various simulator traps. A file is opened when the corresponding inode is created, which happens either during a lookup operation or when a directory is created. The file remains open until the corresponding inode is no longer needed (normally when the vnode is removed or reused).

10.4.2 HostFS VFS Operations

files: ufs/hostfs/hostfs_vfsops.c

Currently, not all file system operations are supported by the host file system. In particular, mounting the file system other than as root has not been tested, and quota operations are not implemented.

The routine *hostfs_mountroot* is used to mount the host file system as root. This routine allocates a root mount point from the VFS layer, mounts the file system and initializes the *statvfs* structure of the mount point (this also copies relevant fields to the old *statfs* structure). Mounting the file system itself involves allocating a file system structure, opening the root directory of the simulation host and copying the root inode and device number into the file system structure.

The *statvfs* routine calls the corresponding simulator trap with the root file descriptor as argument, and then copies the relevant fields into the old *statfs* structure of the file system mount point. The *sync* routine does not need to perform any operations since the host file system does not utilize the buffer cache.

The inode management routine *hostfs_vget* is used to allocate and initialize a new inode/vnode pair. It takes as arguments the file system mount point, the device and inode number and a mode bit field. The routine first checks if the desired inode is already in the hash table and returns a pointer to the corresponding vnode if the inode is already open. Otherwise, it allocates a new vnode from the VFS layer and then allocates a new inode structure. The vnode data field is set to point to the new inode, and the inode is initialized with the device/inode number and mode field. Note that the file descriptor is not set at this point, this has to be done at the caller since this routine has no access to the original file name.

10.4.3 HostFS VNode Operations

files: ufs/hostfs/hostfs_vnops.c

Many of the vnode operations are derived from the FFS code. Modifications are made for file and directory creation and deletion routines. Files and directories are created by calling the simulator trap routine with the parent directory file descriptor and relative path name of the file to be created. The new host file descriptor is stored in the new inode. Similarly, the *stat* and *statvfs* routines call the corresponding simulator trap and copy the result to the appropriate data structures. Note that some routines may fail unexpectedly due to restricted access permission on the simulation host. For instance, some files may be only readable by the superuser, but the Lamix kernel may attempt to access these since it is working under the assumption that the kernel has root privileges. Error handling for these cases is not fully tested. The BSD *readdir* routines have been modified slightly to be compatible with the System V *getdents* calls. All file system vnode routines return 64 bit values where applicable (file offset, file size), which may be converted to the 32 bit representation by the vnode layer.

10.4.4 HostFS Name Lookup

files: ufs/hostfs/hostfs_lookup.c

The *HostFS lookup* routine follows the same basic outline as the original FFS routine. It first checks if the directory where the lookup is to be performed is readable by the process. If this is the case, it performs a lookup in the name cache. If the path name is found in the cache, the corresponding vnode is returned. If no cache entry was found, the routine needs to check the disk for the desired name. However, lookups in the */dev/* directory must not be directed to the simulation host file system, since most device files are not readable by normal user processes, and will not correspond to simulated devices. Instead, the lookup routine detects this case and scans the internal list of devices for the desired name. If a matching entry is found, a vnode is allocated and initialized with the device number and mode bits. For all other directories, the *lookup* routine calls the simulator trap *getdents* and scans the names in the directory structure for a match. If a matching entry is found, the corresponding file is opened and the file descriptor is noted in the new inode.

10.4.5 Device Directory

files: machine/slashdev.c, ufs/hostfs/hostfs_lookup.c
machine/slashdev.h

As discussed above, lookups in the */dev/* directory are redirected to an internal array of supported devices. Each array entry consists of the device name, device type specifier, the device number (major and minor) and a mode bit mask. This table is used to convert device names to device numbers and vice versa (the latter is not used at this point). The lookup routine scans this table for a match when a process performs a lookup in the */dev/* directory. It uses the device number and mode bits to initialize the vnode and inode upon a match.

Entries in the device name table are created dynamically so that the kernel is able to support a large variety of system configurations without the overhead of a large table that contains all possible device names. When creating a new entry, the kernel checks if the name already exists. If it does, and the device number and permission mask is identical, the request is ignored, otherwise the existing entry is overwritten with the new values and a warning is issued.

10.5 BSD Fast Filesystem

files: ufs/ufs/..., ufs/ffs/...

The Berkeley fast filesystem code has been ported directly from NetBSD. The only modifications involves are the update of several VFS routines to reflect the new interface required by the HostFS. This affects routines for vnode lookups (*ffs_vget*), as the HostFS version requires a device number in addition to the inode number as argument, as well as *ffs_readdir* that has been modified to return a Solaris *dirent* structure.

10.6 BSD Log-structured Filesystem

files: ufs/lfs/...

Similarly to FFS, the log-structured filesystem has been ported directly and only with minor modifications from NetBSD. In addition to the same modifications as for FFS, the system call arguments for LFS-specific system calls has been changed to match the Lamix system call convention.

LFS requires that a cleaner daemon performs garbage collection at periodic intervals, or when the filesystem is nearly full. This cleaner daemon is implemented as a separate process running with root privileges. It communicates with LFS through four LFS-specific system calls. For each LFS filesystem mounted, the kernel creates an entry in a list of deferred daemons. Each list entry contains the list of arguments for the daemon, including the full executable pathname, and the process ID of the daemon once it is running. After mounting all filesystems and opening the standard input and output files, but before starting the first user process, the kernel traverses the list to start all required daemon processes.

The LFS cleaner daemon takes as arguments the device name and filesystem mount point. It opens a special *ifile* in the LFS filesystem root directory to read information about the filesystem status. If the filesystem needs cleaning, it performs any required segment management and then blocks in a *segwait* system call. This system call unblocks the daemon when the filesystem is written, or when the specified timeout expires. After waking up, the daemon again checks if the filesystem needs cleaning, performs any necessary management and blocks. Cleaning activity may be deferred if the filesystem is currently busy.

By default, the kernel expects a program called *lfs_cleaner* in the *lamix/* directory, but an alternative name or path may be specified with the *-lfs_cleanerd* kernel command line option.

11 Networking

Lamix includes partial support for networking with Unix and Internet sockets, based on NetBSD source code. The networking subsystem employs a fairly simple object oriented design and consists of three basic components: System Calls, Sockets, Protocols.

11.1 Networking System Calls

files: kernel/uipc_syscalls.c, kernel/socket.c, compat/common/uipc_syscalls_43.c, compat/svr4/svr4_socket.c
sys/socket.h, sys/socketvar.h, compat/svr4/svr4_socket.h

The system calls primarily perform error checking, and then pass on control to the socket level. In some instances, such as *sys_socket*, the system calls perform a bit more work, such as allocating space and initializing the socket. Sockets appear to user programs as file descriptors. The file descriptor structure points to a socket descriptor which includes pointers to the protocol descriptor and pointers to send and receive routines. In addition, each socket descriptor may point to a protocol-specific data structure containing protocol state. The generic file descriptor routines such as *open*, *close*, *read* and *write* are basically wrappers around equivalent socket routines.

11.2 Socket Routines

files: kernel/uipc_socket.c, kernel/uipc_socket2.c, kernel/uipc_mbuf.c
sys/mbuf.h, sys/protosw.h

The socket level functions perform the protocol-independent part of most socket operations. The *socreate* routine determines the desired protocol, allocates a socket descriptor and initializes it. As part of the initialization, it calls a protocol specific routine. Other routines such as *solisten*, *soaccept* and *soconnect* perform only error checking and then call a protocol specific routine. Socket-level routines communicate with lower levels almost exclusively through *mbufs*. Mbufs are structures combining control information, linked-list pointers and a small amount of data. Alternatively, an mbuf may point to external storage such as a malloc'ed buffer. Data is converted between the file system *uio* structure and *mbuf* chains by the socket-level routines. Other arguments such as socket addresses are passed as typecast mbuf pointers, or as mbufs themselves.

Each socket descriptor contains send and receive structures that control transmission and reception of data. These structures provide linked list pointers for mbufs that hold transmit or receive data, variables indicating the amount of data in these lists and the number of mbufs involved, and upper and lower watermarks that control protocol behavior.

When transmitting data with any of the *write* or *send* system calls, the *sosend* routine converts the user data into an mbuf chain, appends it to the sockets transmit structure and calls the protocol send routine.

For incoming data, it is expected that the input routines append similar mbuf chains to the sockets receive structure and update the counter variables appropriately. *Receive* or *read* system calls use *sorecv* to check the availability of data and copy it into user space if possible. Only after removing the mbufs from the socket is a protocol receive routine called to adjust flow control state and possible receive additional data.

11.3 Protocol Routines

files: sys/mbuf.h, sys/protosw.h

Sockets are initialized with a protocol. The *protosw* structure includes function pointers for several functions such as input, output, control and user request, as well as flags that describe protocol details. Protocols may implement four different classes of functions. *output* functions are called synchronously by the upper layer to perform protocol operation in response to system calls or other process activity. *input* functions are called asynchronously by lower layers, either lower-level protocols or device interrupt handlers. These functions handle receipt of data and control information and pass it on to the upper layers.

Most synchronous actions that are initiated by upper levels are executed by the user request function. This function takes an argument that determines which action to perform. This argument corresponds to the higher-level action, such as creating a socket, connecting to a peer or transmitting data. Control information is passed to the protocol through a separate function which handles mostly socket options destined for a particular level.

Protocols do most of the work. They hold the code for interacting with the network or file system on a level that the system calls and sockets do not care about. Lamix currently only supports the *UNIX / LOCAL* protocol and a modified *INET* protocol.

It should be noted that the definition of the socket address structure differs between Solaris/Lamix and BSD. The BSD version uses an 8 bit value for the address family identifier, and another 8 bit value to specify the address structure length itself. Solaris, on the other hand, uses a 16 bit value for the address family. As a result, the size and memory layout of the two structures is otherwise identical. Furthermore, BSD uses the length field only internally, it is first set whenever a socket address is copied from user space into the kernel.

11.4 Unix Protocol

files: kernel/uipc_domain.c, kernel/uipc_usrreq.c
sys/un.h, sys/unpcb.h

This protocol can only be used to communicate between processes on the same machine. It is specified as *UNIX* or *LOCAL* socket domain when creating a socket. Unix domain sockets are identified by file names, which allows the protocol to find peer sockets by using existing name lookup routines. The *unp_bind* routine (the *UNIX* protocol function for bind) calls *VOP_CREATE* to create a vnode and file name that is visible to the user in the file system. A *unp_connect* call from another socket will attempt to access the socket created by *unp_bind* using this name. If the name exists, the vnode structure is used to find the peer socket descriptor.

Reads and writes are handled by exchanging lists of mbufs between the two socket descriptors. When sending data, the mbuf chain created by the socket layer is appended to the peer sockets receive structure, from where it is removed by the receive routine. The Unix protocol does not support datagram sockets, nor does it support out-of-band transmissions.

11.5 Internet Protocol

files: netinet/inh_proto.c, netinet/inh_usrreq.c
netinet/inh.h, netinet/inh_var.h

Lamix provides only rudimentary support for the Internet protocol as a means to establish communication between simulated programs and peers executing outside the simulator. The *inh* protocol is a significantly modified version of the original *INET* protocol which passes most socket operations to the host operating system via simulator traps.

The protocol-specific data structure associated with each internet socket contains a host file descriptor which corresponds to the socket file descriptor visible to the simulator executable. The file descriptor is first established when the socket is created by the *inh_attach* routine. In addition, the socket is marked as *ISCONFIRMING* to force the socket level to call the receive routine before attempting to copy received data.

Bind and listen requests are passed to the simulator via simulator traps and use the host file descriptor to identify the socket. Similarly, accept and connect requests are passed to the simulator. It is important to note that the accept request blocks once it reaches the simulator executable system call level, thus halting all simulation until a process connects to the socket. As a result, the simulation will deadlock if two simulated processes running in the same simulator process attempt to establish an internet domain socket. Also note that since the Internet protocol is not fully simulated, the timing is incorrect. This protocol is mainly useful to provide for communication between simulated and natively-executing processes such as databases or web servers.

Transmit data is shaped into a *msghdr* structure and passed to a *sendmsg* simulator trap. In the process, the protocol transmit routine allocates a page aligned buffer, copies the data from the mbuf chain into the buffer and copies the destination address (if present) from an mbuf into a socket address structure. Upper watermark settings for the socket limit the amount of data transmitted with each request to less then a page size, thus avoiding problems in the simulator trap handler if memory buffers cross a page boundary.

Normally, the socket layer checks the sockets receive buffer for available data, copies it into user space and then informs the protocol layer. To force the socket layer to call the protocol receive routine before checking for data, each *INH* socket is permanently marked as *ISCONFIRMING*. This flag indicates that the socket is in the process of confirming a connection request, and as a result the socket layer calls the protocol receive routine before checking for available receive data. The *INH* receive routine allocates a page aligned receive buffer, assembles a msghdr structure and traps to the simulator with a *recvmsg* trap. This call eventually blocks inside the simulator executable, halting all simulation. To avoid deadlocks, simulated processes should only communicate with processes running outside the simulator. Once data is received into the allocated receive buffer, the buffer is attached to an mbuf as external storage and the mbuf, plus an optional socket address, is attached to the sockets receive structure.

Socket option routines are handled by a separate control output routine. Before calling the protocol specific routine, the socket layer notes all socket options in the socket structure. The *INH* control output routine forwards all socket option changes to the simulation host. Socket option inquiries are handled completely by the socket layer and are not forwarded to the protocol.

12 I/O Subsystem and Device Drivers

This section discusses device drivers and other I/O related portions of the kernel. The disk device driver and related drivers are ported from the NetBSD source code with only minor modifications to adapt them to the Lamix process structure and utility routines provided by Lamix. The realtime clock device driver, however, is structured differently than the BSD based device drivers, in particular it lacks portability and autoconfiguration support.

The following figure shows a logical view of the device drivers and the corresponding hardware.

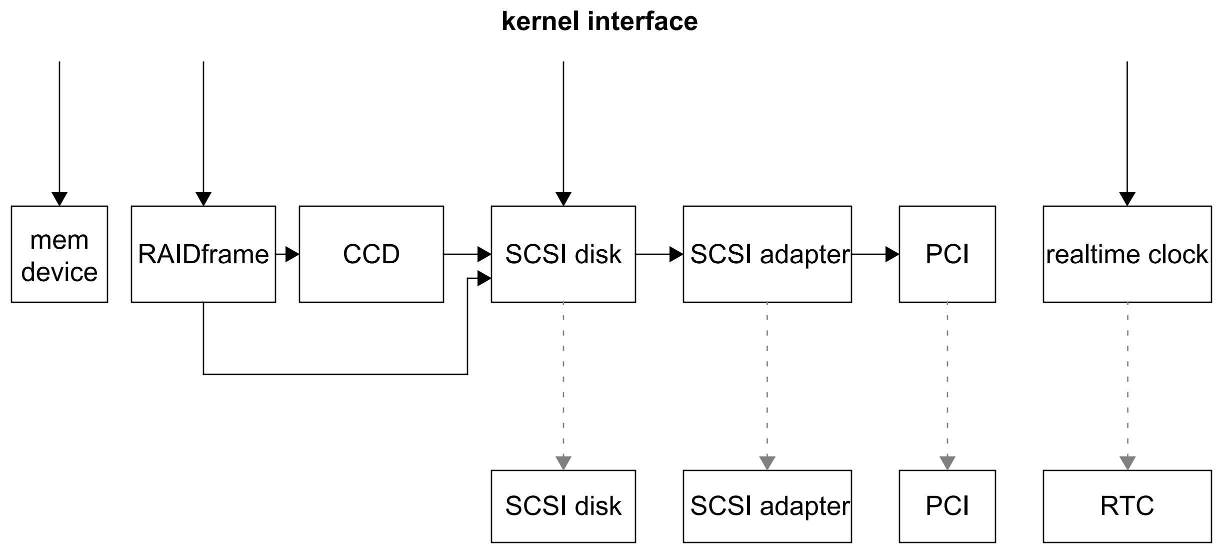

Figure 23: Relationship of Device Drivers and Hardware

The memory device is a pseudo-device, it has no corresponding hardware. The file system layer accesses persistent storage through the RAIDFrame software RAID pseudo device, a simple concatenated pseudo-device or through the SCSI disk driver, mostly by using the respective *Open/Close* and *Read/Write/Strategy* routines.

RAIDFrame is a flexible software RAID device driver that combines any collection of block devices (including other RAID devices) to a RAID set of level 0, 1, 3 or 5. RAIDFrame is a pseudo device, which means it does not directly operate on hardware, it used the block device driver interface of its component devices to function.

The concatenated device driver (CCD) combines multiple disk devices into a larger device, either by simply concatenating the devices, or by striping. Its interface to upper kernel subsystems is identical to a disk, and it can be used as part of a RAID set.

The SCSI disk driver implements the block and raw interface routines for all SCSI disks. The *SDRead/SDWrite* and *SDStrategy* routines assemble generic SCSI transfer control structures that describe the requests and pass them to the SCSI adapter. The SCSI adapter driver converts the request into a device-specific control structure and sets up the request at the device by calling PCI bus routines for reading and writing data or setting up DMA transfers.

Currently, the realtime clock is not connected to the PCI bus and the clock driver communicates with the device directly.

12.1 Realtime Clock

12.1.1 Setup

files: *devices/realtime_clock.c*
devices/realtime_clock.h, interrupts/interrupts.h

Setting up the realtime clock involves installing the page table mapping, and setting up the periodic interrupt. The routine *kernel_rtc_init* installs the mapping for a single page in kernel mode. The page is marked as uncached support.

The initialization routine then initializes the kernels time-variable by reading the clock chips time registers and converting the date structure into the standard Unix format (seconds since 1970). The conversion algorithm has been taken from the NetBSD source code. Note that the access to the realtime clock hardware is protected by the *time_lock* synchronization variable.

The interrupt mechanism is set up by installing the timer interrupt routine in the processors trap table, and turning on the periodic interrupt in the realtime clock chip. Currently, interrupt 1 is configured to interrupt processor 0 every milliseconds. The interrupt vector is 0x0F, which is the highest priority interrupt in the processor.

12.1.2 Clock Interrupt Handler

files: *devices/realtime_clock.c*
devices/realtime_clock.h, kernel/kernel.h, kernel/process.h

The periodic clock interrupt (every 1 ms), is handled only by processor 0. When receiving the clock interrupt, the handler increments both tick counters in the kernel structure. If counter 0 has reached 1000, the routine resets the counter and increments the time variable by 1. Tick counter 1 is used to forward scheduler interrupts to other CPUs. At every tenth interrupt (every 10 ms) CPU 0 increments the global tick counter and calls the *rtc_handler_slave* routine itself to perform clock tick related processing. Other interrupts (tick equals 1 through N) are forwarded to the other CPUs in the system. This means that clock tick processing is offset by 1 ms increments between all CPUs in order to avoid contention for the process list and other kernel structures.

The per-CPU clock tick routine (*rtc_handler_slave*) first determines if the interrupted process was executing in user or kernel mode, and updates the appropriate CPU time counter (*utime* or *stime*) of the calling process. Note that these fields are not protected by a lock because only one CPU at a time can be inside the clock tick handler code. If the process was in user mode, it also decrements the virtual timer if one is set (*it_virt_value* is not 0). If the current timer value reaches 0, it delivers the *VTALRM* signal to the process and restarts the timer by loading *it_virt_curr* with the *it_virt_incr*. If no timer interval is specified (*it_virt_incr* is 0), the timer is disabled by clearing *it_virt_value*. Regardless of the previous processing mode (user or kernel), the routine processes the profile timer in a similar way.

After this, the routine decrements the current processes tick counter. This counter is set to the number of ticks per time slice when the process starts executing in its time slice. If the counter reaches 0, the time slice has been used up and the *need_reschedule* flag is set. This flag is checked when the system returns to user mode, it triggers a context switch when it is set.

Finally, the routine acquires the timer lock and searches through the global timer list for expired timers. For each such timer it first removes the timer from the list and then calls the associated function. In case of the realtime timer, the function either reinserts the descriptor with a new expiration time, or returns it to the free list. After this, the interrupt handler releases the timer-lock and returns.

12.2 Memory Device

files: dev/mem/mem.c

The memory device implements various pseudo-devices that are accessible through the */dev/* directory. The minor device number indicates the subtype of the device, where 0 is */dev/mem* (all physical memory), 1 is */dev/kmem* (kernel memory), 2 is */dev/null* and 12 is */dev/zero*. The two memory subtypes are currently not supported.

The *mmopen* and *mmclose* routines are called when a process opens or closes the device. As the device drive implements very simple pseudo device without any internal state, these routines do nothing. The *mmrw* routine is called for read and write accesses to the character device, its action depends on the minor device number (device subtype). For the */dev/null* subtype, it simply sets the residual count for the read or write operation to zero, indicating that all data was written, or no data can be read (end of file). For writes to */dev/zero*, it also sets the residual count to zero and returns, while for reads it copies the contents of an empty page to the destination address.

The *memattach* routine is called during pseudo-device configuration, it installs several device name entries in the */dev/* directory.

12.3 PCI Driver

files: machine/pci_machdep.c, machine/pio.s, machine/bus_dma.c
dev/pci/pcivar.h

The PCI bus driver is responsible for three machine dependent classes of operations. During configuration, it must provide routines that read PCI configuration space registers. The routine *pci_make_tag* converts a device and function number into an absolute base address for a specific PCI bus. The returned base address is then passed to *pci_conf_read* or *pci_conf_write* together with a register number to perform an uncached read or write to the configuration register.

The *outX* and *inX* routines are used to access individual addresses at the PCI bus. Various routines are provided for different transfer sizes from bytes to words, as well as for streams of data. Note that for stores, these routines must issue memory barrier instructions to flush the write buffer.

DMA operations on the PCI bus are defined by a set of architecture-specific routines in a *bus_dma_tag* structure. These routines are used to create, delete, synchronize, load or unload *dma_map* structures. A *dma_map* is essentially a set of segments that describe contiguous memory regions, along with some flags and other variables. A *dma_map* is created by calling *_bus_dmamap_create*, this routine simply allocates a new map structure and initializes it. The map is loaded with valid mappings in the *_bus_dmamap_load* routine. This routine takes a virtual address and region size as arguments and inserts the corresponding physical address region(s) into the segment list. Note that a contiguous virtual address region can result in one or more physical segments, depending on the physical page layout. Specialized routines exist for different kernel structures such as mbufs and uio structures. A map is unloaded by setting the mapped size and number of segments to 0. The synchronize routine is empty on the ML-RSIM system, since all DMA operations are hardware-coherent. Another routine allocates DMA-safe memory by first allocating memory in kernel space and then building a DMA map for the memory region.

12.4 Adaptec SCSI Adapter

files: dev/pci/ahc_pci.c, dev/ic/aic7xxx.c
dev/ic/aic7xxxreg.h, dev/ic/aic7xxxvar.h

Like most other device drivers, the Adaptec SCSI adapter device driver is taken with only minor modifications from the NetBSD sources. Since the simulated adapter model exhibits a few unique characteristics, a new device subtype AIC 7890 has been introduced. This is a Wide SCSI Ultra-2 PCI controller with a configurable number of SCBs on-chip, and without support for SCB paging. The following figure shows the key routines and data structures used in the device driver.

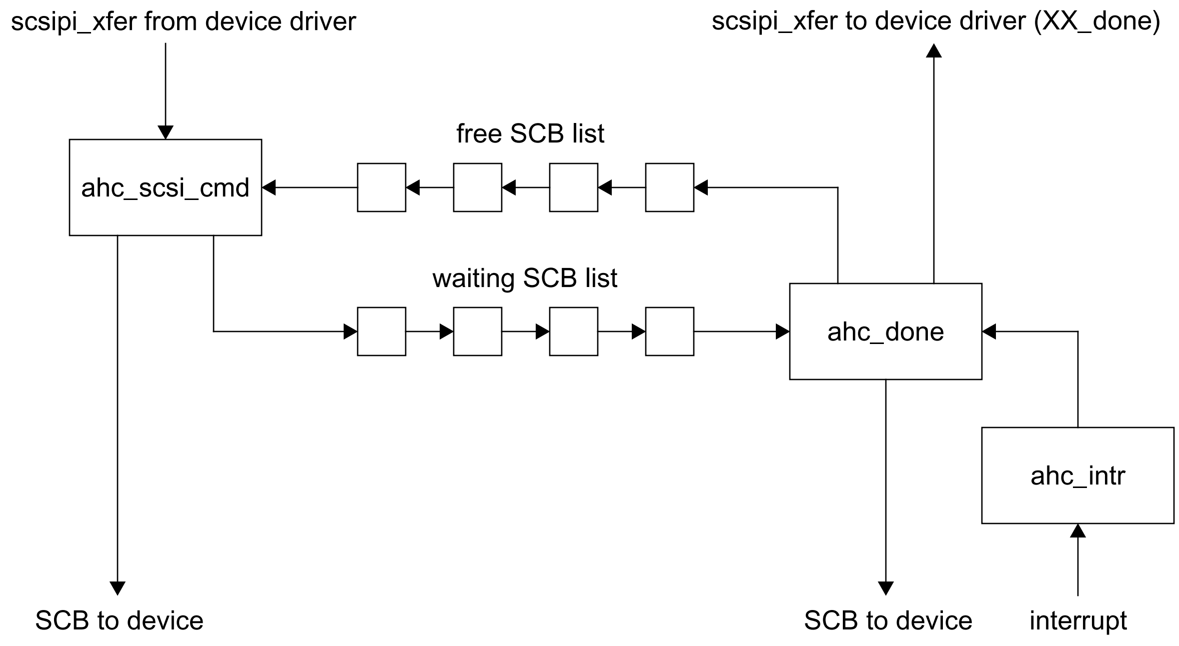

Figure 24: Adaptec Device Driver Structure

12.4.1 Data Structures

The fundamental data structure used in the device driver is an array of SCB structures. Each SCB contains elements that correspond to the hardware SCB array on the device, such as scatter/gather count and scatter/gather list pointer, command and data pointer. In addition, each driver SCB entry provides space for linked-list pointers, an array of scatter/gather segment descriptors as well as space for the SCSI command and for possible return data (such as sense data). The entire array of SCB entries is allocated in DMA-safe memory during initialization. The number of SCBs does not necessarily correspond to the number of SCB entries on the device itself.

SCSI device drivers communicate with the adapter driver through the *scsipi_adapter* and the *scsipi_xfer* structures. This first structure describes the adapter itself, it contains routines to start a SCSI transfer as well as optional routines to enable the device and for *ioctl* calls. Each transfer is described by a *scsipi_xfer* structure. This structure contains various pointers to identify the source and destination of the command, a pointer to the SCSI command and a pointer to the data, as well as space for sense data returned by the device. A SCSI device fills in this structure with the relevant information and passes it to the SCSI adapter driver through the *scsipi_cmd* routine provided by the driver.

12.4.2 Request Processing

The routine *ahc_scsi_cmd* is the main entry point for devices that need the adapter driver to communicate with the device. The routine takes a *scsipi_xfer* structure as argument and converts it into an SCB entry. First, the routine attempts to take an SCB entry off the free-list of SCBs. If no free SCB is found, it either puts the transfer structure in a queue of waiting transfers, or returns with ann error code of queueing is not allowed for this transfer. Once an unused SCB has been found, the routine sets up the SCB elements and copies the command into the SCB command space. If a data transfer is involved, the routine *bus_dmamap_load* is called to translate the virtual buffer address into physical addresses and set up a scatter/gather segment list, which is then copied into the SCB structure. Once the SCB structure is set up, it is either appended to the list of waiting SCBs, or written into an available SCB entry on the device. If the transfer requires polling, the routine then polls on the devices interrupt status register, otherwise it starts a timeout counter and returns. Note that the timeout counter is not used to detect timeout conditions on the SCSI bus, it is only an additional measure to protect the driver from defective devices.

The routine *ahc_done* is called either by the interrupt handler upon command completion or from inside the polling loop. It frees any DMA maps that had been allocated for the data transfer, calls the requesting SCSI devices *done* routine and returns the SCB entry to the free-list.

12.4.3 Interrupt Handler

All SCSI adapter interrupts are handled by the routine *ahc_intr*. It first reads the interrupt status register to determine the class of interrupt.

Sequencer interrupts occur when the sequencer encounters an error such as a rejected message phase or a connect request to an invalid SCB. Some of these events are used for SCB paging, if it is enabled. The most important error is *bad status*, indicating that the device rejected a request due to an invalid command or argument. This happens for instance for read or write requests to an invalid block number. In this case, the driver replaces the command in the current SCB with a *request_sense* command and reissues it to the device. The command is then executed by the adapter and the interrupt handler signals completion of the modified command to the requesting SCSI device. The presence of *sense-data* in the transfer structure indicates that the command did not complete successfully.

SCSI interrupts occur either because of a fatal error on the SCSI bus, such as parity error or unexpected resets, or because of a request timeout. In case of a timeout, the current SCB is removed from the adapter and the timeout condition is marked in the transfer structure.

Command completion is the most common form of interrupts. In this case, the driver removes the SCB index from the *QOUTFIFO* of the device, marks the transfer as done, removes the timeout counter and calls *ahc_done* to signal completion to the requesting device. It then calls *ahc_run_waiting_queue* to move any pending SCBs from the queue of waiting requests to the device and start a new request.

12.5 SCSI Disk

files: dev/scsi/sd_scsi.c, dev/scsi/sd.c
dev/scsi/sdvar.h

The SCSI disk device driver can manage all magnetic or optical disk drives with fixed or removable storage. The driver is selected by the autoconfiguration process for any SCSI device with a matching *inquiry* pattern. When a SCDSI device is successfully detected, the device driver also installs the block and raw device names in the internal device name table.

All SCSI device drivers are described by a *scsipi_device* structure, which contains pointers to routines that start commands, handle interrupts and perform error handling. These routines communicate with the SCSI adapter driver below and the file system layer above. In addition, as both a character and block device, the driver provides routines for opening and closing the device, for reading and writing the character device and a *strategy* routine for accessing the block device. The figure below shows the logical relationship of these routines.

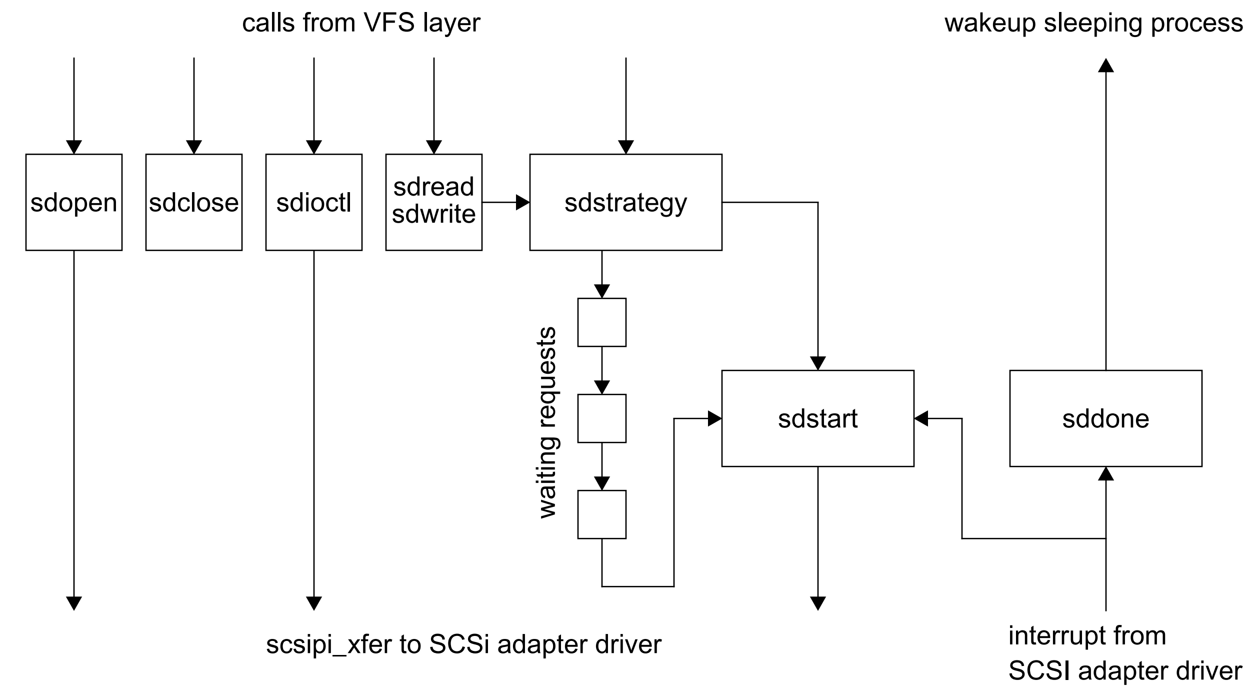

Figure 25: SCSI Disk Driver Structure

The main entry points for the character device are the *open/close* routines and the *read/write* routines. When the character device is opened for the first time, the *sdopen* routine reads the disk label (geometry and partition information) and stores it in an internal buffer for later reference. Additional calls to *sdopen* do not read the label again. When the device is closed for the last time, the driver flushes the disks cache by issuing a *sync_cache* SCSI command, if the disk provides a write cache. The read and write routines utilize the generic *physio* routine which in turn calls *sdstrategy*. The *physio* routine performs character I/O directly to and from user space. It marshals the request parameters into a *uio* structure, pins the physical pages in user space that are involved in the transfer and then calls the *strategy* routine that was passed as parameter.

The block devices main routine is *sdstrategy*, which converts the logical block number for the transfer into a physical number based on the desired partition, inserts the request into the queue of pending requests and calls *sdstart* to send a request to the device if it is not busy. The queue of waiting requests is sorted by block number by the generic *disksort* routine to minimize seek times.

The routine *sdstart* is called either from *sdstrategy* when a new request was added to the queue, or from the SCSI adapter interrupt handler after a request has finished. If the device has space for a new request, it takes the first request off the waiting queue and converts it into a SCSI command that is then passed to the adapter driver. When the request completes, the adapter driver first calls *sddone* to notify the disk driver of the completion. This routine only marks the disk as no longer busy and resets a timestamp. The interrupt handler then calls *sdstart* again to send another request to the device, if one if waiting.

12.6 RAIDFrame

files: dev/raidframe/rf_*.c
dev/raidframe/rf_*.h

RAIDFrame is a software RAID driver based on an implementation from the Parallel data Lab at Carnegie Mellon University. It combines any number of block devices into a RAID set. Component devices may any block device, including other RAID devices as well as SCSI disks. The driver uses directed acyclic graphs to implement RAID levels 0, 1, 3 and 5. Level 0 corresponds to simple striping without redundancy. RAID level 1 mirrors data between disks for maximum redundancy and data protection with highest overhead. RAID level 3 uses parity to reduce the overhead of redundancy, while RAID 5 rotates the parity information over all available disks to avoid the hot spot of a dedicated parity disks. A more detailed description of RAID and the implementation of the RAIDFrame driver is beyond the scope of this document, the reader is referred to the technical report discussing RAIDFrame.

12.6.1 RAIDFrame Request Processing

The RAIDFrame driver, although a pseudo device, exports the same kernel interface as any other block device, including *open, close, read, write, strategy* and *ioctl*. Most of these routines translate requests into a sequence of component device requests using directed acyclic graphs. Acyclic graphs are executed either directly by the calling process, or by a kernel thread. The execution engine normally uses the component device strategy routines to perform sub requests.

Due to its complex nature, the RAIDFrame device implements a large number of *ioctl* commands that are used to configure and unconfigure a RAID set, to initiate parity rewrites and failure recovery and for status inquiries. Long-latency operations such as parity rewrite and reconstruction are non-blocking, the caller may check status periodically using *ioctl* calls. However, the device driver will block other requests until the RAID set is in a stable state.

12.6.2 RAIDFrame Autoconfiguration

The RAIDFrame device is able to automatically configure RAID sets based on the available block devices. Each component of a RAID set is identified by its location in the RAID layout (row & column) as well as a version number. In addition, block device partitions eligible for a RAID set are marked as containing a RAID file system.

After completing the I/O device autoconfiguration, the kernel calls the *raidinit* routine which allocates basic data structures for a number of possible RAID device drivers. It then scans all existing block devices, opens each in turn and reads the partition label. For each RAID partition, it attempts to access the component label to record the component identification information. At the end of the scan, the routine attempts to combine the components into a number of coherent RAID sets based on each components geometry information and version number. For each successfully configured RAID set, the device driver installs the block and raw device names into the internal device name list (see *kernel/slashdev.c*).

12.7 Concatenated Device Driver CCD

files: dev/ccd/ccd.c, dev/ccd/ccdconfig.c
dev/ccd/ccdvar.h

The concatenated device combines multiple disks into a large logical disk, either by simply concatenating the disk partitions, or by striping across all disks of a CCD set. The code base of the CCD device is considerably simpler than the RAIDframe driver, making the CCD device more efficient if simple striping without failure resilience is desired. The device’ *strategy* routine takes requests and produces a set of component requests based on the striping factor. Once all component requests are complete, the device driver signals completion of the request to the upper layer.

Unlike RAIDframe, the CCD device does not support autoconfiguration per se. Instead, when the pseudo-device is attached, the device driver scans all existing disks for a partition with the *CCD* file system type. If one is found, it is assumed that CCD devices should be configured, and a deferred kernel thread is scheduled. This thread (actually only a subroutine) is called after the root file system is mounted and a current working directory is established. To configure one or more CCD sets, the configuration routine performs the same operations as the external *ccdconfig* tool. It reads a configuration file that specifies the CCD device name, interleave factor and method, plus a list of block devices that form the CCD set. The file may contain multiple entries for multiple CCD sets. The default configuration file name is ccd.conf, but other names can be specified via the *-ccdconf=NAME* kernel command line argument. For each line in the configuration file, the configuration routine checks that the component devices are valid and configures a CCD set using an *ioctl* system call.

When a CCD set is configured, the routine *ccdinit* stores the list of component devices, computes the interleave table and creates device entries in the */dev/* directory for each possible partition of the newly configured CCD device. It also registers a shutdown callback routine which closes the component devices and releases all allocated memory.

13 Autoconfiguration

Autoconfiguration is a process by which the kernel builds a hierarchical structure of devices which represents the current system configuration. Each device provides a set of call back functions that are used to configure it and possibly detect any devices that are logically below this device. The kernel provides a set of utility routines to register devices and scan the list of possible devices. The list of supported devices is described by a configuration file which is used by the *config* utility to create an *ioconf.c* file which contains an array of devices descriptors. Each device in the array has a set of possible parent devices. During the configuration process, a device might be detected multiple times, in which case multiple instances of the device are inserted into the device tree. In the tree, each device has exactly one parent a possibly many children. For instance, the *mainbus* is often the root device, with one or more CPUs as well as a number of PCI buses as children. Each PCI bus has a set of controllers as its children, which in turn might have other devices (such as disks) attached.

Note that devices in this context include processors, system buses and bus bridges, which normally are not associated with a device driver. The device hierarchy that is constructed by the autoconfiguration process represents the physical configuration of the system.

13.1 Autoconfiguration Utility Routines

files: kernel/subr_autoconf.c

The routines *config_search* and *config_rootsearch* are used to find matching devices. *config_search* takes as arguments a matching function, a parent device structure and some device-specific data and searches the list of devices for possible children of this parent, applying each devices match-function in turn. If a device is detected, it returns a pointer to the device descriptor.

config_rootsearch performs the same function except that it takes no parent device but a root-device name and it searches for devices that match the specified name. It is used to configure the root device of a system, usually the mainbus.

The routine *config_found_sm* is called when a device has been found (for instance the PCI bus attach routine has found a PCI device), but not been configured. It calls *config_search* with a pointer to a parent-specific match function in order to find the correct device descriptor and the attaches the device.

The routine *config_attach* is called whenever a device has been detected. It takes as arguments a parent device pointer, a pointer to the device descriptor and auxiliary device-specific data. It forms a unique device name, usually by concatenating the generic device name with a unit number, allocates a device control structure and inserts the device in the device tree. It then calls the device-specific attach-function which is found in the device descriptor.

13.2 General Device Configuration Routines

Each device is expected to export a match-function and an attach-function, which are linked into the device descriptor. The match function is called when a possible parent device is configured in order to determine if the device in question is in fact attached to the parent. If this is the case, the match-function returns 1 (true), otherwise it returns 0.

The attach function is called after the device has been detected. It performs any necessary device setup, such as mapping the device registers and assigning interrupts. If the device can have other devices attached to it, it may call *config_found_sm* in order to configure its children.

13.3 Mainbus Configuration

files: machine/mainbus.c

In the Lamix kernel, the mainbus is the device root. It is always present, hence its match-function always returns 1. Furthermore, every mainbus has at least one CPU and a PCI bus attached to it. CPUs are configured by setting the child device name to *cpu* and calling *config_found* for as many times as there are CPUs in the system (as determined by the system control registers). After configuring the CPUs, the routine initializes the PCI bus, assembles a PCI bus device descriptor and attempts to configure the PCI bus by calling *config_found*.

13.4 CPU Configuration

files: machine/cpu.c

Since the mainbus attach routine configures only as many CPUs are there are in the system, no special device detection is done in the CPU match routine, it always returns 1. Configuring a CPU (*cpu_attach*) is done by mapping the respective global system control registers into kernel virtual memory. The local system control page has been mapped in the *init* routine, since access to it is needed very early during the boot process.

13.5 PCI Bus Configuration

files: machine/pci.c, dev/pci/pci.c, dev/pci/pci_subr.c, /dev/pci/pci_map.c
dev/pci/pcivar.h, dev/pci/pcireg.h, dev/pci/pcitypes.h, dev/pci/pcidevs.h

PCI bus configuration is performed in two steps. The architecture-specific initialization includes mapping the PCI configuration space into kernel space and assigning address spaces to all present devices and functions as well as interrupts. The architecture-independent part matches a particular device with its device driver and configures the device driver.

13.5.1 Machine-specific PCI Configuration

files: machine/pci.c

The routine *pci_init* is called by the mainbus configuration routine before the PCI bus itself is configured. This routine first maps the PCI configuration space into main memory and then determines a suitable memory and I/O space address map for the devices that are present.

The I/O address space is split into three segments. The first segment is the PCI memory region, it starts at the base of the IO segment and is 64 MByte big. The PCI I/O region starts at *I/O-base* + 64 MByte and is 32 MByte large. It is followed by a general (non-PCI) I/O device region of 32 MByte. The PCI configuration space is mapped at the beginning of the general I/O segment. It contains an array of 64 256-byte PCI device configuration structures. This allows for a maximum of 8 PCI devices with 8 functions each.

The PCI initialization routine then allocates two arrays with 64 entries describing the requires address spaces. It probes every possible PCI slot if a device is attached. If a device is present, it probes all device functions. For every detected function, it reads the required address spaces by first writing a -1 into the address space base register and then reading back the value. The return value indicates if the corresponding address space requires I/O or memory space and its size. If the size is non-zero, the routine inserts the required size and the device, function and register number into the respective array of address spaces.

In addition, it reads the interrupt pin register to determine if the function requires an interrupt line. If this is the case, it assigns an interrupt number and target CPU based on the general device class. For instance, storage devices are assigned interrupt number 2 on CPU 0. Note that the definition of the interrupt number register does not comply with the PCI standard. The upper 4 bits contain the target CPU and the lower 4 bits contain the actual interrupt number.

After scanning all devices, the *init* routine sorts the collected address-space descriptors by size, and begins assigning consecutive address spaces starting with the largest. This algorithm guarantees that there are no unused regions between address spaces, thus making optimal use of the available PCI space.

13.5.2 Machine-independent PCI Configuration

files: dev/pci/pci.c, dev/pci/pci_subr.c
dev/pci/pcidevs.h, dev/pci/pcivar.h, dev/pci/pcireg.h

After the PCI bus has been configured as described in the previous section, the routine *pciattach* is called to finish configuration. This routine sets up some bus specific data structures and calls *pci_probe_bus* to detect and configure the attached PCI devices.

pci_probe_bus reads the vendor ID register for every possible device slot. If the returned value indicates that the slot is not used (PCI bridge returns -1), the routine skips this slot and continues with the next one. If a device is detected, the routine determines how many internal functions are supported by the device by reading the vendor register for each function. For every successfully detected function, the routine determines the device class interrupt settings, stores these in a per-device data structure and calls *config_found* to configure the device.

Device configuration routines are found by calling all possible PCI device configuration probe functions with the vendor and device ID as parameter. The probe functions return 1 if the device is recognized or 0 if not. When a device is recognized, the autoconfiguration routines call the appropriate attach routine to configure the device. These routines normally set up device specific structures, determine the address range or ranges of the device and possibly perform further autoconfiguration if the device is for instance a SCSI bus adapter.

13.6 SCSI Bus Configuration

files: *dev/scsi/scsiconf.c, dev/scsi/scsipiconf.c*
dev/scsi/scsiconf.h, dev/scsi/scsipiconf.h

The SCSI bus configuration is triggered by calling the routine *scsibusattach* from the device driver of a SCSI adapter. This routine uses adapter specific functions to scan the SCSI bus and determine the type and characteristics of attached devices.

After setting up some data structures, the routine scans all SCSI buses that are attached to the particular adapter by issuing *test_unit_ready* and *inquiry* commands to all possible devices on the bus. If a device responds, the routine then searches for a matching device driver and attaches the device to the internal device tree. The device-specific attach routine normally scans all possibly LUNs and inquires other device specific parameters such as capacity or configuration pages.

13.7 Pseudo-device Configuration

Each pseudo-device provides an *attach* routine which, similar to true physical device drivers, configures and initializes the pseudo-device. These routines normally allocate memory, perform validity checks and install entries in the */dev/* directory.

Each pseudo-device is represented by a structure consisting of a pointer to an *attach* routine plus a maximum device count used during configuration. During startup, the kernel calls each *attach* routine from the list of pseudo-devices.

Part VI: Libraries

Although the simulation system is able to execute unmodified Sparc/Solaris binaries, occasionally special libraries are required to give applications access to special Lamix features, or to provide static libraries not available in Solaris.

1 LibSocket

files: lib_src/socket.S

The Lamix distribution includes its own socket libraries because Solaris does not support static linking for applications that use sockets, but the simulator requires statically linked binaries. The library provides most commonly used socket routines and issues system calls in the same way as the native Solaris library. Executables linked with the Lamix socket library can execute on native Solaris machines as well as on the simulator.

Currently, the library supports the following system calls:

- socket, socketpair
- bind
- listen
- accept
- connect
- getsockopt, setsockopt
- shutdown
- send, sendto, sendmsg
- recv, recvfrom, recvmsg
- getpeername, getsockname

Host name lookups require complex functionality that is normally implemented in several dynamic libraries and which requires access to several system files not available (or meaningful) in the simulator. The socket library provides Lamix-specific versions of the *gethostbyname* and *gethostbyaddr* routines to allow applications compiled and statically linked for Lamix to perform dynamic host lookups. Since these routines rely on a Lamix-specific system call and simulator trap, they can not execute on native Solaris hosts.

2 LibPosix4

files: lib_src/posix4.S

This library implements access to the system calls *clock_gettime*, *clock_settime* and *clock_getres*. It is required because Solaris provides only a dynamic version of this library, which is not supported by Lamix. Since the actual system call wrapper is implemented in the standard C library, *libposix4* implements only the entry points and branches back to the C library.

3 LibUserStat

files: syscall/userstat.c, syscall/userstat_asm.S

This library implements wrapper code for the user statistics collection mechanism of Lamix. It provides the system call *userstat_get* to allocate and name a user statistics entity, and the system call *userstat_sample* that adds a sample to an existing statistics entity. This library can be linked in by applications by specifying the *-luserstat* flag at the linker command line.

Part VII: System Administration Tools

The system administration tools described in this section are taken from the NetBSD source. They are unmodified except for cases when they assume the presence of a particular system file (such as */etc/disktab*) which can not be accessed in the simulator. These tools occasionally share source code or include files with the kernel, in which case a symbolic link is made to the kernel sources.

1 Disklabel

files: apps/sbin/src/disklabel.c, apps/sbin/src/dkcksum.c, apps/sbin/src/interact.c
apps/sbin/src/opendisk.c
apps/sbin/src/ext_disklabel.h, apps/sbin/src/disktab.h, apps/sbin/src/dkcksum.h
apps/sbin/src/pathnames.h, apps/sbin/src/paths.h, apps/sbin/src/util.h

Harddisks are identified to the kernel by a disk label which is located at a known block address. The disk label contains information about the drive mechanics and organization (such as seek time, number of cylinders, blocks etc.) as well as the layout of the various partitions. When a new disk is created, the *disklabel* tool must be run first to create a valid label.

The *disklabel* tool allows to read and write disk labels in a variety of ways, and is also able to install a boot image on a disk. However, the simulator version of this tool does not support all of the different parameters. Mainly, the tool is used for two purposes: to write a disk label that is described in an ASCII file, and to read the disk label on a disk. The following command line shows how to use the tool to write a new disk label:

```
mlrsim -F -nomount -root disklabel -r -R /dev/rsd0c <labelfile>
```

The *-r* option forces the tool to update the label on the physical medium rather then the label kept inside the kernel. The second option (*-R*) instructs the tool to write the disk label as it is specified in the label file. The device name must point to a character device that corresponds to a disk, and should also identify partition 2 (c) as the raw partition. The following is an example of a label file that specifies three file system partitions (two FFS partitions and one LFS partition) and a raw partition (number 2) that covers the entire disk.

```
# /dev/rsd0c:
type:                SCSI
disk:                IBM :4296Mb
label:               sim-disk
flags:
bytes/sector:         512
sectors/track:        209
tracks/cylinder:        5
sectors/cylinder:    1045
cylinders:           8420
rpm:                 7200
interleave:             1
trackskew:             50
cylinderskew:          30
headswitch:             0.5     # milliseconds
track-to-track seek:    0.8     # milliseconds
drivedata:              0

4 partitions:
```

```
#     size   offset   fstype  [fsize bsize  cpg]
a: 2933315     1045   4.2BSD   1024  8192   32  # (Cyl.    1 - 2807)
b: 2933315  2934360   4.4LFS                    # (Cyl. 2808 - 5615)
c: 8798900        0    boot                     # (Cyl.    0 - 8419)
d: 2930180  5867675   4.2BSD   1024  8192   32  # (Cyl. 5616 - 8419)
```

Note that the first partition does not start at cylinder 0 because the first few blocks are reserved for the disk label, thus rendering the entire cylinder unusable. Offsets and partition sizes are specified in sectors, but must be a multiple of the cylinder size. *fsize* and *bsize* are parameters used when the file system is created, they specify the fragment size and block size in bytes. *cpg* specifies the number of cylinders per *cylindergroup* for the fast file system.

A disk label can be read by calling the tool without the *-r* option and without a label file. The output can be saved in a file and used to write other disk labels.

In either case the tool should be simulated with the *-nomount* option to prevent mounting of a disk that is to be modified. In addition, the *-root* option is necessary to give the application access to the raw device.

2 NewFS

files: *apps/sbin/src/newfs.c, apps/sbin/src/mkfs.c, apps/sbin/src/dkcksum.c, usr/lib/ufs/ffs/ffs_bswap.c*
apps/sbin/src/ext_newfs.h, apps/sbin/src/dkcksum.h, apps/sbin/src/disktab.h, apps/sbin./src/mntopts.h apps/sbin/src/paths.h, apps/sbin/src/util.h

This utility is used to create a FFS file system on a partition. The file system parameters are derived from the disk label but can be overwritten at the command line. Normally, the only necessary command line parameter is the character device that corresponds to the disk and partition where the file system is to be created. For instance, the following example creates a file system on the second partition (number 1) of disk 0:

```
mlrsim -F -root -nomount newfs /dev/rsd0b
```

The tool first reads the label from the specified disk, determines the file system parameters and checks if they are within reasonable ranges. It then writes the cylinder group summaries for each group, creates the root directory and makes copies of the superblock. Depending on the partition size and number of cylinder groups, this process can take several days when running under the simulator. Note that the access permissions for the root directory are set to allow read, write and execute access for all users, not just the superuser. This allows normal users to create files and directories in the root directory immediately after the file system has been created.

3 NewLFS

files: apps/sbin/src/newlfs.c, apps/sbin/src/lfs.c, apps/sbin/src/misc.c, apps/sbin/src/lfs_cksum.c apps/sbin/src/ext_newlfs.h, apps/sbin/src/config.h, apps/sbin/src/util.h apps/sbin/src/disktab.h

Similar to *newfs,* this tool creates a LFS file system on the specified partition. Normally, the only necessary command line parameter is the character device that corresponds to the partition where the file system is to be created. The following command line shows how to create a file system on partition b of a disk:

```
  mlrsim -F -root -nomount newlfs /dev/rsd0b
```

The tool determines all required file system parameters from the disk label. It then writes the root directory, inodes, the index file and creates copies of the superblock and segment descriptors. Creating a LFS file system is significantly faster then creating a FFS file system. Again, please note that the default permissions of the root directory give all users read, write and execute access the directory.

4 LFS_Cleanerd

files: lfs_cleanerd/cleaner.c, lfs_cleanerd/library.c, lfs_cleanerd/misc.c, lfs_cleanerd/print.c lfs_cleanerd/clean.h

The LFS cleaner is not a conventional system utility but a daemon that is normally started by the kernel when an LFS filesystem is mounted. As such, users should not be required to manually start the cleaner daemon. Several command line arguments are supported to control cleaning thresholds and policies, timeouts and to specify the filesystem to clean. The following command line is currently used by the kernel to invoke the daemon:

```
lfs_cleaner -t 60 <fs_name> <device>
```

The first argument lowers the timeout interval to 60 seconds from the default 5 minute value. The last two arguments are required and specify the mount point of the LFS filesystem and the special block device for the filesystem.

Unless specified with the *-lfs_cleanerd* kernel command line parameter, the *lfs_cleaner* executable should be located in the same directory as the kernel image used for a simulation run.

5 FSCK

files: *apps/sbin/src/fsck.c, apps/sbin/src/preen.c, apps/sbin/src/fsutil.c*
apps/sbin/src/main.c, apps/sbin/src/dir.c, apps/sbin/src/inode.c
apps/sbin/src/ffs_main.c, apps/sbin/src/ffs_dir.c apps/sbin/src/ffs_inode.c
apps/sbin/src/ffs_setup.c, apps/sbin/src/ffs_utilities.c apps/sbin/src/ffs_pass1.c
apps/sbin/src/ffs_pass1b.c, apps/sbin/src/ffs_pass2.c, apps/sbin/src/ffs_pass3.c
apps/sbin/src/ffs_pass4.c, apps/sbin/src/ffs_pass5.c, usr/lib/ufs/ffs/ffs_tables.c
usr/lib/ufs/ffs/ffs_subr.c, usr/lib/ufs/ffs/ffs_bswap.c
apps/sbin/src/lfs_main.c, apps/sbin/src/lfs_dir.c, apps/sbin/src/lfs_inode.c
apps/sbin/src/lfs_setup, apps/sbin/src/lfs_utilities, apps/sbin/src/lfs_vars.c
apps/sbin/src/lfs_pass0.c, apps/sbin/src/lfs_pass1.c, apps/sbin/src/lfs_pass2.c
apps/sbin/src/lfs_pass3.c, apps/sbin/src/lfs_pass4.c, usr/lib/ufs/lfs/lfs_cksum.c
apps/sbin/src/ext_fsck.h, apps/sbin/src/fstab.h, apps/sbin/src/fsutil.h
apps/sbin/src/paths.h, apps/sbin/src/pathnames.h, apps/sbin/src/ffs_extern.h
apps/sbin/src/ffs_fsck.h, apps/sbin/src/lfs_extern.h, apps/sbin/src/lfs_fsck.h

The *fsck* utility checks the consistency of a file system and performs any necessary repair. This is useful if the simulator or the simulated application crashed while a file system was mounted.

The tool actually consists of a front end which checks the file system type and parameters, and different backends for various file system types. The front end opens the raw device, reads the disk label and determines the file system type for the desired partition. It then forks itself and the child calls the backend executable which performs the actual file system check and repair. The following command performs a file system check on partition a of the first disk.

```
mlrsim -F -root -nomount fsck -l 1 -y /dev/rsd0a
```

Note that the parameter *-l 1* instructs the tool to run only one *fsck* backend at a time. The *-y* parameter indicates that the answer to all inquiries by the backend is *yes*, which is important since the kernel and simulator currently do not support interactive user input.

Currently, backends exist for the FFS and LFS file systems. Before starting the detailed file system check process, the backends check if the file system is clean (has been unmounted properly) in which case they exit, or dirty. Both backends operate in various phases, checking the superblocks, free lists, inode tables and so on. To improve performance, the backends maintain a cache of disk blocks. Nevertheless, a complete file system check simulates for several days, depending on the partition size.

6 RaidCtl

files: apps/sbin/src/raidctl.c, apps/sbin/src/rf_configure.c, apps/sbin/src/opendisk.c apps/sbin/src/raidframevar.h, apps/sbin/src/raidframeio.h, apps/sbin/src/rf_configure.h

The *raidctl* utility is used to configure and maintain the RaidFRAME software RAID device driver. While the RaidFRAME driver supports autoconfiguration and detects drive faults, manual intervention is needed for the initial setup of a RAID set. The following table lists the most common flags and modes of operation.

Parameter	Description
-A yes/no <dev>	enable/disable autoconfiguration for the raid set
-A root <dev>	enable autoconfiguration and make device eligble for a root filesystem
-c <file> <dev>	configure the device according to the configuration file
-C <file> <dev>	force configuration of the device according to the configuration file
-i <dev>	initialize device, in particular rewrite parity
-I <serial> <dev>	initialize component labels and write serial numbers for autoconfigurayion
-p <dev>	check device status
-P <dev>	check status and rewrite parity
-u <dev>	unconfigure device
-v	verbose output

The *raidctl* tool provides a multitude of other flags for adding and removing spare disks, performing failure recovery and diagnostic. The table above lists only the most common operations.

The following sequence of steps is needed to set up a new raid device:

- write disk labels
- configure RAID set (use -C to force configuration)
- enable autoconfiguration if desired (recommended)
- initialize component labels
- rewrite parity
- create disk label for RAID device
- create file systems

For autoconfiguration to work, partitions used to form a RAID set should be marked as containing a RAID file system. The configuration file contains at least four sections, describing the overall RAID geometry, which devices make up a RAID set, the RAID layout and queue sizes and policies. Each section begins with the keyword *START* and the section name. Section *array* lists the number of rows (usually one) and columns and spare disks. The *disks* section lists the component devices making up the RAID set. These devices may be disk partitions or RAID devices themselves. Similarly, the optional *spares* section lists devices used as spares. The *layout* section describes the RAID layout in terms of blocks per stripe unit, stripe units per parity units, stripe units per reconstruction unit and RAID level. Finally, the *queue* section specifies a queuing method and queue size to be used by the device driver. The following sample configuration file defines a RAID level 5 set with 4 components.

```
START array
# numRow numCol numSpare
1 4 0

START disks
/dev/sd0a
/dev/sd1a
/dev/sd2a
/dev/sd3a

START layout
# sectPerSU SUsPerParityUnit SUsPerReconUnit RAID_level_5
16 1 1 5

START queue
fifo 100
```

Since the *raidctl* command requires access to the raw RAIDFrame device, it must be run with root privileges and without mounting any of the disks involved in the RAID set operation, i.e.

```
mlrsim -F -nomount -root raidctl -C raid5.conf raid0
```

Rewriting parity and several other operations take a considerable amount of time, as the entire RAID set must be scanned and possible reconstructed. The *ioctl* calls to initiate these operations are nonblocking, the raidctl tool periodically checks progress via another *ioctl* call and prints status information. Since output is currently always written to a regular file, the status output appears as a long list of lines instead of a constantly updated line of text.

7 CCDConfig

files: apps/sbin/src/ccdconfig.c, apps/sbin/src/fparseln.c

The *ccdconfig* utility provides a runtime user-interface to configure and unconfigure sets of concatenated disks, and to perform other diagnostic operations. The following table lists the supported command line flags

Parameter	Description
-c	configure a CCD, default
-C	configure all CCD devices listed in the configuration file
-f config_file	use the specified file instead of the default (/etc/ccd.conf)
-g	dump current CCD configuration
-M core	extract values for dumping from ‘core’ instead of /dev/mem
-N system	extract list of CCD names from ‘system’ instead of /kernel
-u	unconfigure a CCD device
-U	unconfigure all CCD devices listed in the configuration file
-v	verbose output

The most common use of the tool is to configure one or all CCD sets. Since the Lamix kernel imports the simulation host file system, an alternate configuration file name should always be specified, otherwise the utility would attempt to open the default file. The dump flag is currently not supported, since it requires access to the memory special device which is not implemented in Lamix. Note that the kernel attempts to configure CCD devices semi-automatically by performing essentially the same functions as *ccdconfig* if at least one disk partition was labeled as *CCD*. However, the CCD device driver itself does not require components to be marked as *CCD,* hence this tool can be used to concatenate other devices.

The configuration file consists of a list of devices, one per line. Each line specifies the CCD device name, the interleave factor and method and a list of component devices, as shown in the example below:

```
# ccd    ileave     flags           component devices
ccd0     16         none            /dev/sd0a /dev/sd1a /dev/sd2a
ccd1     0          none            sd3b sd4b
ccd2     32         CCDF_UNIFORM    /dev/sd5f /dev/sd6a /dev/sd7d
```

The first and third entry specify an interleaved CCD set, while the second entry results in a simple concatenated device. The interleave factor is given in component block size (usually 512 bytes). If the *CCDF_UNIFORM* flag is specified, the interleaving is uniform, even if components have different sizes. This leads to wasted space on larger components. Component names can be given as absolute path, or using just the device name.

References

[1] Adaptec, Inc. *AIC-7770 Data Book.* 1992.

[2] Rémy Card, Éric Dumas, Franck Mével. *The LINUX Kernel Book.* John Wiley & Sons. 1998.

[3] William V. Courtright II, Garth Gibson, Mark Holland, LeAnn Neal Reilly, Jim Zelenka. *RAIDFrame: A Rapid Prototyping Tool for RAID Systems*, Version 1.0, Technical Report, Carnegie Mellon University, Pittsburgh, PA, 1996.

[4] Gregory R. Ganger, Bruce L. Worthington, and Yale N. Patt. *The DiskSim Simulation Environment Version 1.0 Reference Manual*. Technical Report CSE-TR-358-98. Department of Electrical Engineering and Computer Science, University of Michigan, February 1998.

[5] Edward K. Lee. Performance Modeling and Analysis of Disk Arryas. Ph. D. Dissertation, Technical Report CSD-93-770, University of California, Berkeley, 1993.

[6] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, John S. Quarterman. *The Design and Implementation of the 4.4. BSD Operating System.* Addison Wesley Longman, Inc. 1996.

[7] MIPS Technologies. *MIPS R10000 Microprocessor User's Manual*, Version 2.0, 1996.

[8] Vijay S. Paj, Parthasarathy Ranganathan and Sarita V. Adve. *RSIM Reference Manual*, Version 1.0. Technical Report 9705. Department of Electrical and Computer Engineering, Rice University. August 1997.

[9] ST Microelectronics. *M48T02 Data Sheet.* November 1998.

[10] Sun Microsystems. *UltraSPARC User's Manual.* 1997.

[11] David L. Weaver, Tom Germond. *The SPARC Architecture Manual Version 9*. PTR Prentice Hall Inc. 1994.

Appendix A: Directory Structure

```
 ml-rsim
    apps                       generic makefile, sample applications
                               and empty disk files
         batch_shell           simple batch processing utility
              src              source code
              obj              object files
              execs            Solaris/Lamix executables
              outputs          directory for simulation output
         bin                   utilities, e.g. ls, rm, mkdir ...
              src ...          same directories as above
         dirtest               directory-related system call tests
              src ...          same directories as above
         exec                  exec system call test
              src ...          same directories as above
         fac                   tests fork, shmem, wait
              src ...          same directories as above
         filetest              file I/O system call tests
              src ...          same directories as above
         long                  long-running simple computation
              src ...          same directories as above
         longf                 long-running FP computation
              src ...          same directories as above
         sbin                  other utilities, e.g. raidctl, newfs
              src ...          same directories as above
         short                 short meaningless computation
              src ...          same directories as above
         signals               signal system call tests
              src ...          same directories as above
         socket_test           UNIX and INET domain socket tests
              src ...          same directories as above
         sysconf               prints various system config. info.
              src ...          same directories as above
         time                  time system call tests
              src ...          same directories as above
         ccdtest               scripts & config. files for CCD device
              outputs          directory for simulation output
         raidtest              scripts & config. files for RAID device
              outputs          directory for simulation output
```

```
bin                        contains simulator Makefile
     IRIX                  architecture specific binaries
     IRIX64
     Linux
     SunOS
doc                        documentation
lamix                      Lamix kernel source code and binary
     arch                  architecture-specific kernel code
          lamix_ws         Lamix code & configuration
               conf        configuration file and utility
               lamix_ws    machine-specific code
     compat                compatibility code
          common           common code
          svr4             System V compatibility code
     conf                  global configuration directory
     dev                   device drivers
          ccd              concatenated device driver
          ic               chip drivers (AHC 7xxx)
          mem              memory pseudo device
          pci              pci drivers
          raidframe        software RAID driver
          rtc              realtime clock
          scsipi           SCSI bus and device drivers
          sun              Sun-specific files
     fs                    global filesystem code
     interrupts            interrupt handlers and definitions
     kernel                general kernel code
     lfs_cleanerd          LFS cleaner deamon
     lib_src               user library source code
     libkern               kernel library sources
     miscfs                misc. filesystems
     mm                    memory management
     net                   general networking code
     netinet               Internet networking code
     sys                   include files
     syscall               general system call code
     ufs                   common filesystems
          ext2fs           Linux ext2 FS, incl. file req'd by FFS
          ffs              fast filesystem
          hostfs           Lamix host filesystem
          lfs              log-structured FS (not fully supported)
          ufs              general UFS sources
```

```
sim_mounts                  mountpoints for simulated disks
     sim_mount0
     sim_mount1
     etc...
src                         ml-rsim source code and object files
     sim_main               utility routines
          Objs              platform-specific object files
               IRIX         (these will not be listed for the
               IRIX64        following directories)
               Linux
               SunOS
     Bus                    system bus model
          Objs              platform-specific object files
     Caches                 cache models, incl. uncached buffer
          Objs              platform-specific object files
     DRAM                   DRAM backend models
          Objs              platform-specific object files
     IO                     IO models, incl. disk & realtime clock
          Objs              platform-specific object files
     Memory                 memory controller model
          Objs              platform-specific object files
     Processor              CPU model
          Objs              platform-specific object files
```

Appendix B: Summary of Parameters

Parameter	Description	Default
numnodes	number of nodes	1
numcpus	number of processors per node	1
kernel	Lamix kernel filename	../../lamix/lamix
memory	size of memory, affects only file cache size	512M
clkperiod	CPU clock period in picoseconds	5000
activelist	number of active instruction, ROB size	64
fetchqueue	size of fetch queue/instruction buffer	8
fetchrate	instructions fetched per cycle	4
decoderate	instructions decoded per cycle	4
graduationrate	instructions graduated per cycle	4
flushrate	instructions flushed per cycle after except.	4
maxaluops	maximum number of pending ALU instructions	16
maxfpuops	maximum number of pending FPU instructions	16
maxmemops	max. number of pending memory instructions	16
shadowmappers	number of unresolved branches	8
bpbtype	type of branch predictor (2bit,agree,static)	2bit
bpbsize	size of branch predictor buffer	512
rassize	size of return address stack	4
latint	integer instruction latency	2
latshift	integer shift latency	1
latmul	integer multiply latency	3
latdiv	integer divide latency	9
latflt	floating point operation latency	3
latfconv	FP conversion latency	4
latfmov	FP move latency	1
latfdiv	FP divide latency	10
latfsqrt	FP sqare root latency	10
repint	integer instruction repeat rate	1
repshift	integer shift repeat rate	1
repmul	integer multiply repeat rate	1

Table 27: Summary of Simulator Parameters

Parameter	Description	Default
repdiv	integer divide repeat rate	1
repflt	FP instruction repeat rate	1
repfconv	FP conversion repeat rate	2
repfmov	FP move repeat rate	1
repfdiv	FP divide repeat rate	6
repfsqrt	FP suare root repeat rate	6
numaddrs	number of address generation units	1
numalus	number of integer functional units	2
numfpus	number of FP functional units	2
storebuffer	size of processor store buffer	16
dtlbtype	data TLB type (direct,set_assoc,fully_assoc)	direct
dtlbsize	data TLB size	128
dtlbassoc	data TLB associativity	1
itlbtype	instr.TLB type (direct,set_assoc,fully_assoc)	direct
itlbsize	instr. TLB size	128
itlbassoc	instr. TLB associativity	1
cache_frequency	frequency relative to CPU core	1
cache_collect_stats	collect statistics	1
cache_mshr_coal	max. number of misses coalesced into a MSHR	8
L1IC_perfect	perfect L1 I-cache (100% hit rate)	0 (off)
L1IC_prefetch	L1 I-cache prefetches next line on miss	0 (off)
L1IC_size	L1 I-cache cache size in kbytes	32
L1IC_assoc	L1 I-Cache associativity	1
L1IC_line_size	L1 I-cache line size in bytes	32
L1IC_ports	number of L1 I-cache ports	1
L1IC_tag_latency	L1 I-cache access latency	1
L1IC_tag_repeat	L1 I-cache access repeat rate	1
L1IC_mshr	L1 I-cache miss status holding register size	8
L1DC_perfect	perfect L1 D-cache (100% hit rate)	0 (off)
L1DC_prefetch	L1 D-cache prefetches next line on miss	0 (off)
L1DC_writeback	L1 D-cache writeback	1 (on)
L1DC_wbuf_size	size of L1 D-cache write bufer	8

Table 27: Summary of Simulator Parameters

Parameter	Description	Default
L1DC_size	L1 D-cache cache size in kbytes	32
L1DC_assoc	L1 D-Cache associativity	1
L1DC_line_size	L1 D-cache line size in bytes	32
L1DC_ports	number of L1 D-cache ports	1
L1DC_tag_latency	L1 D-cache access latency	1
L1DC_tag_repeat	L1 D-cache access repeat rate	1
L1DC_mshr	L1 D-cache miss status holding register size	8
L2C_perfect	perfect L2 cache	0 (off)
L2C_prefetch	L2 cache prefetches next line on miss	0 (off)
L2C_size	L2 cache size in kbytes	512
L2C_assoc	L2 cache associativity	4
L2C_line_size	L2 cache line size	128
L2C_ports	number of L2 cache ports	1
L2C_tag_latency	L2 cache tag access delay	3
L2C_tag_repeat	L2 cache tag access repeat rate	1
L2C_data_latency	L2 cache data access delay	5
L2C_data_repeat	L2 cache data access repeat rate	1
L2C_mshr	L2 cache miss status holding register size	8
ubuftype	combining or nocombining buffer	comb
ubufsize	number of uncache buffer entries	8
ubufflush	threshold to flush uncached buffer	1
ubufentrysize	size of uncached buffer entry in 32 bit words	8
bus_frequency	bus frequency relative to CPU core	1
bus_width	bus width in bytes	8
bus_arbdelay	arbitration delay in cycles	1
bus_turnaround	number of turnaround cycles	1
bus_mindelay	minimum delay between start of transactions	0
bus_critical	enable critical-word-first transfer	1 (on)
bus_total_requests	number of outstanding split-transaction reqs.	8
bus_cpu_requests	number of outstanding CPU requests (per CPU)	4
bus_io_requests	number of outstanding I/O requests (per I/O)	4

Table 27: Summary of Simulator Parameters

Parameter	Description	Default
io_latency	latency of I/O device bus interface including PCI bridge and bus	1
numscsi	number of SCSI controllers per node	1
ahc_scbs	number of control blocks on Adaptec cntrl.	32
scsi_frequency	SCSI bus frequency in MHz	10
scsi_width	SCSI bus width in bytes	2
scsi_arb_delay	SCSI bus arbitration delay in bus cycles	24
scsi_bus_free	minimum SCSI bus free time in cycles	8
scsi_req_delay	lumped delay to transfer a request in cycles	13
scsi_timeout	SCSI bus timeout in cycles	10000
numdisks	number disks per SCSI bus	1
disk_params	disk parameter file name	<none>
disk_name	name of disk model	IBM/Ultrastar_9LP
disk_seek_single	single-track seek time	0.7
disk_seek_av	average seek time	6.5
disk_seek_full	full stroke seek time	14.0
disk_seek_method	method to model seek time (disk_seek_none, disk_seek_const, disk_seek_line, disk_seek_curve)	disk_seek_curve
disk_write_settle	write settle time	1.3
disk_head_switch	head switch time	0.85
disk_cntl_ov	controller overhead in microseconds	40
disk_rpm	rotational speed	7200
disk_cyl	number of cylinders	8420
disk_heads	number of heads	10
disk_sect	number of sectors per track	209
disk_cylinder_skew	cylinder skew in sectors	20
disk_track_skew	track skew in sectors	35
disk_request_q	request queue size	32
disk_response_q	response queue size	32
disk_cache_size	disk cache size in kbytes	1024
disk_cache_seg	number of cache segments	16
disk_cache_write_seg	number of write segments	2

Table 27: Summary of Simulator Parameters

Parameter	Description	Default
disk_prefetch	enable prefetching	1 (on)
disk_fast_write	enable fast writes	0 (off)
disk_buffer_full	buffer full ratio to disconnect	0.75
disk_buffer_empty	buffer empty ratio to reconnect	0.75
mmc_sim_on	enable detailed memory simulation	1 (on)
mmc_latency	fixed latency if detailed sim. is turned off	20
mmc_frequency	memory controller frequency relative to CPU	1
mmc_debug	enable debugging output	0 (off)
mmc_collect_stats	collect statistics	1 (on)
mmc_writebacks	number of buffered writebacks	numcpus + number of coherent I/Os
dram_sim_on	enable detailed DRAM simulation	1 (on)
dram_latency	fixed latency if detailed sim. is turned off	18
dram_frequency	DRAM frequency relative to CPU	1
dram_scheduler	enable detailed DRAM timing	1 (on)
dram_debug	enable debug output	0 (off)
dram_collect_stats	collect statistics	1 (on)
dram_trace_on	enable trace collection	0 (off)
dram_trace_max	set upper limit on number of trace items	0 (no limit)
dram_trace_file	name of trace file	dram_trace
dram_num_smcs	num. data buffers/multiplexers & data busses	4
dram_num_jetways	number of data buffers/multiplexers	2
dram_num_banks	number of physical DRAM banks	16
dram_banks_per_chip	number of chip-internal banks	2
dram_rd_busses	number of data busses	4
dram_sa_bus_cycles	number of cycles of an address bus transfer	1
dram_sd_bus_cycles	number of cycles of a data bus item transfer	1
dram_sd_bus_width	width of data bus in bits	32
dram_critical_word	enable critical-word-first transfer	1 (on)
dram_bank_depth	size of request queue in SMC	16
dram_interleaving	block/cacheline and cont/modulo	0 (cacheline modulo)
dram_max_bwaiters	number of outstanding requests	256

Table 27: Summary of Simulator Parameters

Parameter	Description	Default
dram_hotrow_policy	open-row policy	0
dram_width	width of DRAM chip = width of DRAM data bus	16
dram_mini_access	minimum DRAM access size	16
dram_block_size	block interleaving size	128
dram_type	type of DRAM (SDRAM or RDRAM)	SDRAM
sdram_tCCD	CAS to CAS delay	1
sdram_tRRD	bank to bank delay	2
sdram_tRP	precharge time	3
sdram_tRAS	RAS latency, row access time	7
sdram_tRCD	RAS to CAS delay	3
sdram_tAA	CAS latency, column access time	3
sdram_tDAL	data-in to precharge time	5
sdram_tDPL	data-in to active time	2
sdram_tPACKET	number of cycles to transfer one 'packet'	1
sdram_row_size	size of an open row in bytes	512
sdram_row_hold_time	maximum time to keep row open	750000
sdram_refresh_delay	number of cycles for one refresh	2048
sdram_refresh_period	refresh period in cycles	750000
rdram_tRC	delay between ACT commands	28
rdram_tRR	delay between RD commands	8
rdram_tRP	delay between PRER and ACT command	8
rdram_tCBUB1	read to write command delay	4
rdram_tCBUB2	write to read command delay	8
rdram_tRCD	RAS to CAS delay	7
rdram_tCAC	CAS delay (ACT to data-out)	8
rdram_tCWD	CAS to write delay	6
rdram_tPACKET	number of cycles to transfer one packet	4
rdram_row_size	size of an open row in bytes	512
rdram_row_hold_time	maximum time to keep row open	750000
rdram_refresh_delay	number of cycles for one refresh	2048
rdram_refresh_period	refresh periods in cycles	750000

Table 27: Summary of Simulator Parameters