Paint:
PA Instruction Set Interpreter [1]

Leigh B. Stoller
Mark R. Swanson
Ravindra Kuramkote
E-mail: {stoller,swanson,kuramkot}@cs.utah.edu
WWW: http://www.cs.utah.edu/projects/avalanche

UUCS-96-009

Department of Computer Science
University of Utah
Salt Lake City, UT 84112, USA

September 11, 1996

Abstract

This document describes **Paint**, an instruction set simulator based on Mint[3]. Paint interprets the PA-RISC instruction set, and has been extended to support the Avalanche Scalable Computing Project[2]. These extensions include a new process model that allows multiple programs to be run on each processor and the ability to model both kernel and user code on each processor. In addition, a new address space model more accurately detects when a program is accessing an illegal virtual address, allows a program's virtual address space to grow dynamically, and does lazy allocation of physical pages as programs need them.

Note that this document is intended to be an addendum to the original Mint technical report, which the reader should consult for an overview of the Mint simulation environment and terminology.

[1]This work was supported by a grant from Hewlett-Packard, and by the Space and Naval Warfare Systems Command (SPAWAR) and Advanced Research Projects Agency (ARPA), Communication and Memory Architectures for Scalable Parallel Computing, ARPA order #B990 under SPAWAR contract #N00039-95-C-0018

Contents

1 Introduction

This note describes the Paint (PA Interpreter) simulation environment. Paint is based on the Mint[3] simulation system developed at the University of Rochester, and has been modified to interpret the PA-RISC[1] instruction set and to support the Avalanche Scalable Computing Project[2]. These changes are documented here. The reader is encouraged to read the original Mint report before proceeding, but as a review the next few sections present the essential concepts. More detailed descriptions follow later.

1.1 Program Driven Simulation

Paint is a program-driven simulator, partitioned into two main parts: a memory reference generator (the "frontend") and a target system simulator (the "backend"). The frontend models the execution of a program by simulating the instruction stream. When an instruction causes or requires special operation, such as a memory reference or special system instruction, the frontend generates an event for the backend to operate on. The backend models the memory hierarchy and interconnect of the target system, including, but not limited to, the first level cache, the TLB, the system bus, main memory, and the network interface. When the operations for carrying out the event have completed, the backend signals the frontend to continue execution of the instruction stream for that processor.

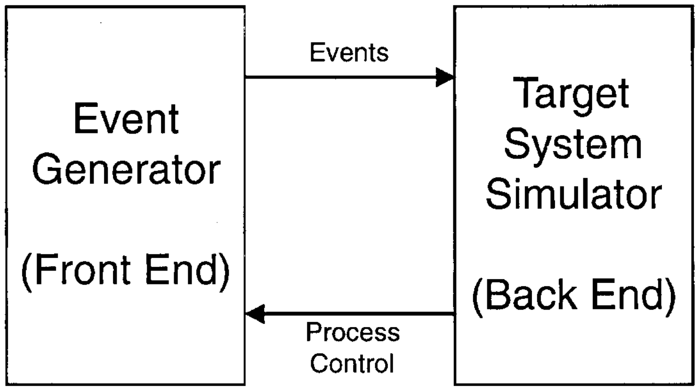

Figure 1: Program Driven Organization

Program execution in Paint is interpreted; the instruction stream consists of a sequence of data structures representing the actual instructions. The state of the processor is represented in a global data structure. Processor state includes the values of registers, virtual to physical page translation tables, the current program counter, etc. As instructions are interpreted, the value of this structure changes. When the simulator switches to a new processor, a different global structure is installed as the current processor, and execution continues as before.

When a program is loaded, the text portion of the file is scanned and converted to a linked list of structures called *instruction codes* (or *icodes*). The icodes are linked together, for both sequential and non-sequential execution (branches and jumps). Each icode stores information about how to interpret the instruction, as well as a pointer to a function to handle the actual interpretation. In general, each instruction has a specific function, although some have more than one when certain opportunities for optimization are detected. Execution then consists of calling the function for each instruction, which may modify the processor state, and which returns a pointer to the next instruction icode to simulate, which might be the next sequential instruction or the target of a

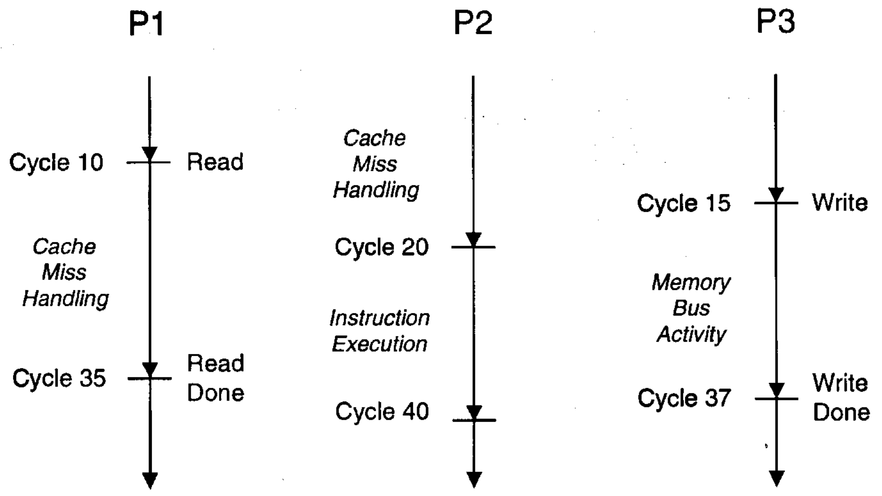

Figure 2: Execution Timeline

branch or jump.

1.2 Virtual Memory Model

Memory reference instructions are handled specially by the Paint frontend. Once the user virtual address is computed from information in the icode structure, a virtual to physical translation must be performed. Two translations are actually produced; the first is a "processor" physical address that is used by the backend, and the second is a "paint" physical address that corresponds to the actual location within Paint's address space. The processor physical address is used by backend modules that require realistic physical addresses, like a cache or memory bus module. The Paint physical address is used as the location to actually read and write data to memory. When the TLB module is in use, the simulated kernel sets the processor physical address using appropriate target machine instructions. If the TLB is switched off, the processor physical address is set equal to the Paint physical address.

1.3 BackEnd Interface

As mentioned above, the simulator frontend is responsible for executing instructions until something interesting occurs, such as a memory reference. At this point instruction execution is suspended and an *event* is generated for the backend. The event is a data structure that packages up details about the event so that they may be communicated to the backend. The backend then operates on the event, possibly scheduling *tasks* to handle event activities. Tasks are scheduling entities that contain a time to run and a dispatch function to invoke when the specified time arrives. In this way, multiple concurrent activities can be in progress, including: instruction execution by processors that are not blocked waiting for an event to complete, memory hierarchy activities in support of processor load and store instructions, and network transmission and reception. When the event is complete, the backend signals the frontend that instruction execution for the specified processor may proceed (in fact, instruction execution is a task associated with each processor). See figure 2.

2 Process Model

By far, the most widely reaching change to Mint was to the *process model.* Mint was originally a Single Program, Multiple Data (SPMD) system. A program would start, possibly do some initialization, and fork one or more children. Each child was considered not a only a new process, but a new processor as well. In fact, process and processor were essentially the same. The disadvantage of this model is that multiple programs cannot be run, forcing a particular programming model that is not always appropriate for distributed memory machines. Further, by not being able to simulate multiple programs per node, the time and memory effects of "kernel mode" cannot be measured since all operating system functionality was implemented in the simulator itself. This has the effect of making many operations cost free, thus skewing the simulation results.

The process extensions made to Mint allow it to run a functional kernel on each node (see section 9.2), and an arbitrary set of user programs on each node. Operationally, each simulated processor is represented by a single Paint thread. The kernel is the first program to run within that thread, followed by 1 or more user level programs. Like a real machine, the kernel context switches bewteen user programs with appropriate target machine instructions that change the register state of the processor. Paint maintains an association between the different programs and the simulated processor those programs are running on, which allows Paint to switch the instruction stream and virtual address context when requested to do so by the kernel.

Before presenting a detailed description of the frontend operation, a user level view of the process model is given.

2.1 User Level View

When Paint is started, the program it is given to simulate is the kernel. The kernel then duplicates itself on each virtual processor as the first program using the **newvproc()** system call. Once the kernel is running on the new processor, it forks a child, and execs the *init* program. The init program reads a setup file that specifies which programs to run on each processor, and forks/execs the programs for its processor. The init program then exits. At this point the kernel goes into an *idle* loop, waiting for system calls and interrupts. Of course, the kernel does not actually idle since executing instructions that do nothing is too costly in a simulated environment. Instead, the kernel puts itself to sleep. The simulator wakes the kernel up when it needs to do something.

Since the goal is to run "user" program binaries unchanged, and without special compilation, the system call interface is identical to the one used in the BSD and HPUX kernels. When the simulator detects a system call in a user program, it vectors the instruction stream for that thread to a known address in the kernel. Eventually the kernel must handle the system call. For most calls, it means letting the simulator take care of it (see Section 8 in the Mint User Manual) by "calling" the intended system call function, just as the user program did. The difference is that when the kernel calls a system call (say, open(...)), the simulator intercepts the call and handles the operation, returning a result (a file descriptor in the case of open()). The kernel then returns the value to the user program just as a production kernel does. The goal is to have the kernel catch all system calls so it can decide which ones are handled in the simulated kernel, and which are passed onto the simulator itself.

Multiple programs can be run on each node. Additional support from the simulator allows the kernel to context switch between multiple kernel threads. When the simulator executes one of several PA instructions (*be, ble, rfi*), the simulated instruction stream is switched to a different set of instructions, as defined by the PA architecture. In other words, a single Paint thread multiplexes several simulated kernel threads using real context switch code to change register and program counter values.

Process scheduling is done in the kernel, using the BSD 4.4 scheduling subsystem. The simulator generates simulated clock interrupts that are delivered asynchronously to the kernel so that it may update the scheduling data structures, recompute process priorities, and possibly arrange for the current process to be context switched out. Kernel timers are also supported. The default period of the clock is 100,000 cycles, and is a configurable option to the simulator using the `VPROC_clockperiod` parameter value (see section 6).

Asynchronous interrupts and traps are handled in a manner similar to system calls. When a simulation module generates an interrupt or a trap for a processor, the instruction stream for the currently running process on that processor is vectored to a known location in the kernel. A standard state save is done (written in assembly language), then a call is made to a C dispatch function to handle the interrupt.

The following sections describe the Paint frontend in more detail. Later sections expand further on key areas.

2.2 Instruction Execution

Instruction execution is the most basic operation in Paint. At its simplest, the *instruction loop* takes the current instruction, represented by a pointer to an `icode`, calls the dispatch function contained in the icode, and receives back a pointer to the next instruction to execute. This repeats until an *event* is generated, or until a maximum number of instructions have been executed in a row. At this point a rescheduling operation is performed, and a new *task* is selected to run. This new task might invoke the instruction execution loop for a new processor, or it might be a task that is working on an event for some processor, or it might be an anonymous task that is scheduled to perform some operation in a simulation module. This operation repeats until there are no more tasks scheduled to run, at which time the simulation terminates.

When a task does invoke the instruction loop, it begins execution with the current instruction pointer. Figure 3 shows the icode data structure. Many of the fields are specific to the actual instruction. For example, the `immed` field holds the signed immediate value for any instruction whose format includes an immediate. Other fields have a common usage during execution, and should be described:

`func`	The function to invoke to handle the actual simulation of the instruction.
`next`	A pointer to the next sequential instruction in the code stream.
`target`	A pointer to the branch target instruction when the instruction is a conditional or unconditional branch, and the target can be computed statically.
`cycles`	The number of CPU cycles the instruction consumes, not including memory hierarchy delay. The value is added to a running count as instructions are simulated.
`validregs`	The set of scaler registers used by the instruction, represented as a bitmask, 1 bit for each of 32 registers. This field is used to implement *stall on use* loads (see section 4).
`validfregs`	The set of floating point registers used by the instruction, represented as a bitmask, 1 bit for each of 64 singles, or 2 bits for each of 32 doubles (see section 4).

```
typedef struct icode {
        PFPI        func;
        char        args[4];
        icode_ptr   next;
        icode_ptr   target;
        u_long long validfregs;
        u_long      validregs;
        long        immed;
        u_short     cycles    :5,
                    form_addr :2,
                    cfield    :5,
                    aum       :4;
        char        cp;
        char        s_or_clen_t;
        u_long      f_flag    :1,
                    nullify   :1,
                    next_null :1,
                    is_target :1,
                    opnum     :12,
                    opflags   :16;
        long        addr;
} icode_t, *icode_ptr;
```

Figure 3: Instruction Code Data Structure

Each instruction function returns a pointer to the next icode to execute. The next icode is either the next sequential instruction, or the target of a branch, or a dynamically computed jump target. The first two pointers are computed when the program is loaded, and stored in the icode structure. The pointer to the next icode for a dynamically computed jump target is returned by the `T2I` function, which takes the target address of the jump as its argument. When the instruction loop is suspended, the last icode pointer is stored in the processor data structure (this is effectively the program counter), and the cycle count for the processor is incremented. The instruction execution task for the processor may be rescheduled to run at the new processor cycle count, or it might not if the loop was suspended due to an event. Time moves forward since each processor runs for a short time (possibly ahead of other processors), with all processors eventually getting a chance to move forward. (see figure 2).

2.3 Events

While the Paint frontend is primarily concerned with instruction execution, it is the Paint backend that is responsible for more detailed simulation of selected architectural features. The most obvious example is the memory hierarchy. In the absence of event generation, all loads and stores would take a fixed amount of time, which is unrealistic. The backend writer can instead model a detailed memory hierarchy. Meanwhile, other processors can continue ahead until some synchronizing event occurs. There are many types of events that can be generated for the backend. This document will concern itself with just memory events, so the reader should consult the Mint[3] document for

```
typedef struct task {
        struct task         *next;
        struct task         *prev;
        int                 pricrity;
        int                 pid;
        mint_time_t         time;
        PFTASK              ufunc;
        struct event        *pevent;
        int                 ival1;
        void                *uptr1;
} task_t, *task_ptr;
```

Figure 4: Task Data Structure

a discussion of other events.

Events in the Paint frontend look much like an instruction. They are represented by `icodes`, but with a function pointer to a routine that initiates the event. These special icodes are placed into the instruction stream when the program is loaded. The loader scans the instructions, and in the case of memory reference instructions, creates a duplicate icode, flags it as an event, and replaces the function with the appropriate event routine (either `event_read()` or `event_write`). The original icode is left as the next sequential icode. In other words, each memory reference icode is preceded by a new event icode that causes the frontend to suspend execution and invoke the appropriate backend function. When the backend signals that execution can continue, the original icode is executed to effect the changes in processor state required by the particular instruction. For example, PA-RISC loads and stores do base register modification, which must occur after the memory reference completes.

2.4 Tasks

When the backend function for an event is called, it is given a single argument, a pointer to the task controlling instruction execution for that processor (see figure 4). When the backend function returns, it indicates via a status value whether instruction execution should continue or suspend until some future time, and whether the task should be put back on the free list. If execution is suspended, then it is up to the backend to save and eventually reschedule the task so that instruction execution may proceed. The fields of the `task_t` data structure are:

`next, prev`	Queuing elements. These fields can be used only when the task is not currently scheduled to run since they are also used by the scheduling system.
`time`	The absolute time at which the task should be run.
`priority`	The task priority. If multiple tasks need to run in the same timestep, and they need to be ordered, a priority can be assigned to force one task to run before or after another.
`pid`	The processor ID the task is executing on behalf of.
`ufunc`	The function to invoke when the task runs. This function must return one of:

`T_ADVANCE`	The processor associated with this task may continue executing instructions.
`T_FREE`	This task is put on the free list and the next task with the minimum time is removed from the task queue and executed.
`T_YIELD`	This is the same as `T_FREE` except that the task is not put on the free list. Only a reschedule is performed.

`pevent`	A pointer to the event data structure that was constructed by the frontend.
`ival1,uptr1`	Storage location for arbitrary values to pass to the function. There are many more such variables, so the reader should consult the header file.

When the backend function is invoked, the **`pevent`** field of the **`task_t`** data structure points to the event structure created by the frontend. The event structure is quite large and can accommodate many types of events, so the reader should consult both the Mint document and the **`''event.h''`** header file. The various task scheduling functions are described in the Mint document.

3 Paint Address Space

This section describes the changes to address space translation. Paint dynamically translates addresses during simulation, using a simple address translation formula that converts a program "virtual" address into an address in Paint's "physical" space. There were several characteristics of the original memory model that needed improvement:

- Address space protection: Program errors can easily generate illegal virtual addresses, which when translated to physical addresses, reference data in another program, or in the simulator itself. The translation mechanism should check the validity of each virtual addresses presented for translation.
- Dynamic allocation of memory: The program's data and stack segments should be allowed to extend past their original size as needed.
- Lazy allocation of memory: The physical pages for the bss, heap and stack should not require allocation until they are referenced by the running program. This would reduce the number of unused, and thus wasted, pages. With a small number of nodes, this is not an issue, but 32 and 64 node simulations of even moderate sized programs become difficult, even on machines with hundreds of megabytes of swap space and real memory.

Our approach was to implement page tables in the simulator. Page tables allow us to accurately detect when a program is accessing an illegal virtual address, to implement dynamically sized segments, and to allocate physical pages lazily as the program needs them. An additional benefit is that TLB information can be stored in the page tables. Finally, a recent optimization allows cache line status to be stored in additional data structures attached to each page table entry. By utilizing this information in the frontend, calls to the backend for each and every memory event can be avoided, resulting in a twofold increase in simulator performance.

TLB support was then implemented using fields in the page table structure. Before a memory event is allowed to proceed, the corresponding page table entry is accessed. If the page is marked as currently being in the TLB, and the read/write access permission bits match the type of access, the memory event is allowed to proceed normally. If the page is not in the TLB, or if the access type is wrong (ie: writing a read-only page), a TLB miss is generated by calling `tlb_domiss()` in ``tlb.c.'' In addition to some book keeping, a processor trap is generated with `vproc_trap()` (see section 5.3). Subsequent TLB insertion instructions (for the PA, `idtlbp` and `idtlba`) executed by the kernel cause the TLB information for that page to be updated and the instruction retried. The full cost of the TLB is modeled.

The current TLB model is very simple. A 96 entry, fully associative TLB is modeled by maintaining a list of page table structures that are currently in the TLB. When a capacity miss requires an entry to be replaced, the oldest entry is removed from the list, and the new one entered. The size of the TLB can be altered with the `TLB_numentries` parameter field (see section 6). While the PA-RISC supports a rather rich set of TLB options (protection identifiers, multiple privilege levels, etc.), the simulated TLB supports only read/write access permissions. By default, TLB modeling is turned on in the simulator. To turn the TLB off, use the `TLB_on` parameter field, setting it to 0.

The following subsections describe the specifics of address translation.

3.1 Address Space Organization

Figure 5 shows Paint's address space organization. Process virtual addresses are mapped to both a processor physical address, and to a Paint physical address. The processor physical address is assigned by the simulated kernel. The purpose of this address is to provide realistic processor physical addresses to simulation modules such as the cache or memory bus. Processor physical addresses are supplied by the kernel with PA-RISC TLB insertion instructions. When the TLB is turned off, the processor physical addresses are always set to the Paint physical address. The page table entry (figure 6) includes several TLB bits that indicate if the page table entry is currently in the TLB, and the type of access permissions (read or read/write) that the translation was inserted with.

Paint physical addresses are locations inside of the Paint program where simulated program data is actually stored. These addresses are known only to Paint, and are assigned when a memory reference touches a page for the first time. It is not until this point that a page in Paint's program space is allocated and the page table entry created (see Figure 6). This lazy allocation of pages allows larger simulations since it is often the case that programs never reference many of their pages. This arrangement allows programs to grow more dynamically as well. Like a real kernel, program segments are assigned a maximum size (16 megabytes for data, 2 megabytes for stack), and a vector of page table entry *pointers* for each possible page is created. A program can grow, lazily allocating pages until it reaches that maximum. The overhead is small, given that there are only 4096 page table entry pointers for a 16 megabyte segment, or 16K bytes of storage per segment. Both maximum values can be overridden using the parameter file entries `MAX_STACK_SIZE` and `MAX_DATA_SIZE`. Several other fields in the page table entry should be noted:

`type`	The type of page, currently either a normal data page or a shared memory page. This information is passed to the backend.
`dealloc`	Flag to indicate whether the underlying physical page should be deallocated when the page table is reclaimed. This is used when page table entries share a common physical page (as with shared memory segments).
`hits`	Cache hit information. Used in the frontend to avoid calls to the backend cache module. The frontend and the backend co-manage this data structure. See section 3.4.

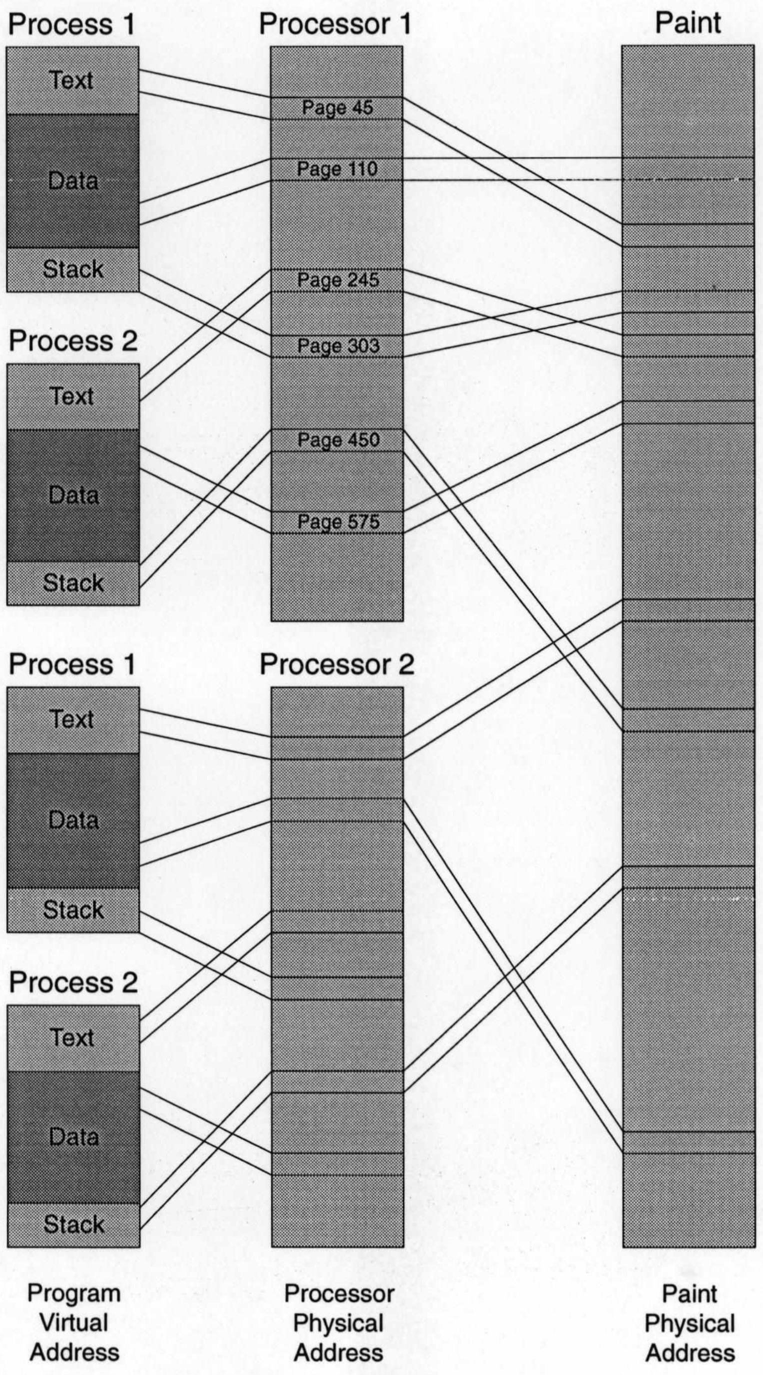

Figure 5: Paint Address Space

```
typedef struct {
        u_long    pframe   :20,  /* Paint Physical Page */
                  type     :12;
        u_long    vframe   :20,  /* Program Virtual Page */
                  pbits    :12;
        u_long    fframe   :20,  /* Processor Physical Page */
                  dealloc  :1,
                  tlbvalid :3,
                  reserve  :8;
        ptcline_t hits[PTCSIZE];
} ptable_entry_t, pte_t;
```

Figure 6: Page Table Data Structure

3.2 Shared Memory Support

Shared memory support is provided in Paint through the use of the page table system. The virtual address range starting at 0xC0000000 is defined to be the shared memory address space. Each processor has a single set of page table entries for the segment, and they are shared among all of the processes on a processor. The underlying Paint physical pages are shared among all of the page table entries on all of the processors. Thus, memory accesses on different processors refer to the same memory location since they share a single page. In the Paint frontend, the only difference when generating the memory event is that it is flagged as being to the shared memory segment. Any special handling is expected to be done in the backend by the cache module.

3.3 Fork and Exec

Paint provides the support necessary for both of the UNIX system calls, `fork` and `exec`. When a user process executes a fork or exec system call, the kernel does whatever bookkeeping it requires, and then passes the call onto Paint itself. In the case of `fork`, Paint then duplicates both the process page tables and the contents of the pages. The virtual page addresses are the same in the child's version of the page table entries, but there are new Paint physical addresses for each duplicated page of data. Pages that had not been touched in the parent, and thus were not allocated, are left unallocated in the child. All of child's page table entries are marked as *not* being in TLB, and the processor physical addresses are cleared. When the process eventually runs, normal TLB misses will provide the new processor physical addresses. This mimics the operation of a real kernel in which `fork` duplicates exactly the virtual address range, but maps those virtual addresses to a new set of physical pages. For `exec`, Paint first reclaims all of the page table entries and pages, and then loads the new program. A new set of page table entries is created, and the initialized data is loaded. All other pages (bss, heap, stack) are allocated lazily as the program references them.

As can be seen, much of the support for `fork` and `exec` is contained within Paint itself. The kernel includes its own support, but is much simpler that a production kernel would be. The bulk of the machine dependent virtual memory support is contained in ``vm.c'' in the kernel.

3.4 Cache Support

The Paint frontend includes several extensions that allow it to optimize memory events. The first is called *fasthits* mode, and is used to decrease the number of memory references that result in backend events. Decreasing the number of backend events improves overall simulator performance since suspending the instruction loop and invoking the backend is a very costly operation. When fasthits mode is turned on (using the `PT_fasthits` parameter field), the frontend consults the `hits` field of the page table entry to determine if the cache line being accessed is currently in the cache. If it is, no backend event is generated, and instruction execution continues immediately. Execution will continue until a maximum threshold of sequential instructions is reached, at which time a rescheduling operation is performed so that a new task may run. The threshold defaults to 50 instructions, and can be set using the `PT_fastcount` parameter value.

The backend cache module is responsible for telling the frontend which lines are currently in the cache. There are two functions provided to the backend, one to indicate that a line has been inserted into the cache and another to indicate that a line has been evicted from the cache. In addition, the backend provides a function to the frontend so that the frontend can signal when a fasthit dirties a cache line. Thus, the frontend and the backend co-manage this state information as the simulation proceeds. The prototypes for the two function called by the backend are:

```
ptable_cache_validate(int procid, int spaceid, unsigned long vaddr);

ptable_cache_invalidate(int procid, int spaceid, unsigned long vaddr);
```

The prototype for the function called by the frontend is:

```
flc_changestate_dirty(int procid, int spaceid, unsigned long vaddr);
```

For each routine, `procid` is the processor number, `spaceid` is the PA-RISC space identifier for the access, and `vaddr` is the virtual address of the line being accessed.

The second optimization mode provided by the frontend is called *fastmisses* mode, and is controlled by the `PT_fastmisses` parameter value. Fastmisses mode requires fasthits mode be turned on. When fastmisses is on, no memory events (except those to I/O space locations) are generated. Instead, the frontend calls a function in the backend cache module to indicate that a line has been accessed. This allows the cache to be *warmed up* with the proper data, but without the expense of going to the backend. This mode is most useful during startup and initialization phases where the speed of the simulation is more important than accuracy. The function prototype provided by the backend is:

```
flc_fastcache_insert(int procid, int spaceid, int pid,
                     unsigned long vaddr, unsigned long paddr, int rw);
```

In order for this mode to be useful, it is necessary to provide a mechanism to turn it off at some point during the simulation, switching to the more accurate cache model. A simulated program level function call is provided that can be used in either the kernel or a user program to turn fastmisses mode on or off. This function is trapped by Paint itself:

```
fastmissmode(int onoff);
```

4 Stall On Use; Asynchronous Writes, I/O Space

This section describes the changes necessary to support *stall on use* (SOU), *asynchronous writes*, and *I/O space access* The PA-RISC cache model employs all of these features, so supporting them was essential for realistic simulations in the Avalanche project. Stall on use allows the processor to proceed after a load operation, until the target of the load is referenced in a subsequent instruction. Only then must the processor stall until the load is complete. Asynchronous writes allow the processor to proceed immediately after a store. In this case, the cache stalls the processor when there are no more slots in which to hold the pending store. I/O space memory references are required to access Avalanche devices that are mapped into regions of the processor's address space.

Supporting these new features required changes to the frontend and to the interface between the frontend and the backend.

4.1 Valid Registers

Stall on use support requires that the frontend know which registers are referenced in each instruction. For non-memory instructions, the source registers must all be valid. The target registers must also be valid before allowing the operation to proceed, even though real hardware would not necessarily require it. This is to prevent prior loads to the same register, that have not completed yet, from subsequently overwriting the target. We do this as a simplification since it rarely happens that a register is destroyed before a previous load to that same register is used (and thus, would stall). For memory instructions, the base and index registers, as well as the target registers must all be valid before the instruction can proceed.

In order to determine which registers are referenced by each instruction, and which registers are currently valid during execution, a **validregs** data structure was added to the **icode_t** structure. When a program binary is loaded, the validregs structure for each instruction is initialized with a list of registers that are referenced in that instruction. A similar structure was also added to the processor structure. As execution proceeds, the current set of validregs for the processor is compared against those referenced in each instruction. If all of the referenced registers are valid, the instruction executes normally. If there are invalid registers, the processor is stalled until the backend indicates to the frontend that execution can continue.

Load instructions are a special case since they modify the current set of valid registers. The target register of the load is made invalid. At some later time, the backend will indicate that the load has completed, and that the register can be made valid again. If the processor had been stalled because of a subsequent reference to that register, it is restarted. This state change is communicated to the frontend using the **sou_load** function described below.

The validregs information for each instruction is initialized using the **SETVALIDREG** macros (there are variants for integer, floating point, and double registers). See the instruction decode functions in `''text.c''` for an example.

4.2 Asynchronous Events

The original interface between the frontend and the backend was through the use of **event_t** structures. This structure carried all of the information needed by the memory system module to carry out the operation. This interface is synchronous in nature; the backend must either stall the processor immediately, or copy all of the information out of the event structure before allowing the processor to continue. This is because there is just one event structure per instruction execution task, which is reused for all events that are sent to the backend. All of the information in the event

```
typedef struct {
        unsigned long           value[2];
        unsigned long           *paddr;
        unsigned long           vaddr;
        short                   regnum;
        short                   type;
        short                   spaceid;
        short                   vproc;
        short                   pid;
} rw_trans_t;
```

Figure 7: rw_trans_t Data Structure

structure must be captured before the frontend is allowed to continue, or it will be lost when the next event is reached. In order to support a more asynchronous interface between the frontend and the backend, a new structure, **rw_trans_t** was introduced. See figure 7. This new structure saves the backend from having to copy out the event information. A brief description of the fields follows:

`value`	The value being written to memory in a store instruction, or the value being loaded in an I/O space load. The value is captured since the operation may not proceed until a later time, and the data might be altered before then by other tasks. The field is large enough to support double word operations.
`paddr`	The Mint "physical" address of the memory location the data is being written to or read from.
`vaddr`	The program "virtual" address that was referenced.
`regnum`	The register number that is the target of a load instruction.
`type`	Various type bits to indicate such things as the size of the operation, whether it is to the shared address space, etc.
`spaceid`	A PA-RISC implementation specific space identifier.
`vproc`	The processor number.
`pid`	The global process identifier.

The above structure is created by the frontend when calling any of the backend functions **sim_read()**, **sim_write()**, **sim_flush()**, **sim_purge()**, or **sim_sync()**. A pointer to the structure is placed in the **pending** slot of the **event_t** structure. The backend function can copy that pointer, but if it plans to let the processor continue asynchronously, it must set the pointer to NULL before returning `T_ADVANCE`. This tells the frontend to create a new structure at the next event. This is an optimization that prevents the creation of a new data structure on each event unless the backend captures the previous one. All other backend event functions use the original **event_t** structure interface as described in the Mint document.

Once the backend determines that a load or store operation can proceed, and the contents of the registers or memory can be changed, it will call either **sou_load** (for load instructions) or

sou_store (for store instructions) to handle the actual operation. This prevents the backend from having to know numerous internal details of the frontend.

4.2.1 sou_load()

```
sou_load(rw_trans_t *ptrans, mint_time_t simtime);
```

sou_load loads the contents of memory into a register. **ptrans** points to a **rw_trans_t** structure captured by **sim_read** at some time in the past. The register is marked valid and data transferred, and if the processor is stalled, it is restarted. **simtime** is the time at which the processor should be restarted. A side effect of **sou_load** is to free the structure pointed to by **ptrans**.

4.2.2 sou_store()

```
sou_store(rw_trans_t *ptrans);
```

sou_store stores the contents of a register to memory. **ptrans** points to a **rw_trans_t** structure captured by **sim_write** at some time in the past. The value of the register at the time the store was done is contained within the **rw_trans_t** structure; it is this value that is stored to memory. A side effect of **sou_store** is to free the structure pointed to by **ptrans**.

4.2.3 Simple Read Example

The following example is a simple demonstration of SOU. It is by no means complete, and is not functionally correct in the presence of asynchronous writes.

```
#include "mint.h"

int read_done(task_ptr ptask);

sim_read(task_ptr ptask)
{
    rw_trans_t *ptrans    = ptask->pevent->pending;
    task_ptr   *pnewtask;

    ptask->pevent->pending = NULL;

    /* schedule the read_done() function 10 cycles from now */
    pnewtask = sched_task(ptask, read_done, ptask->time + 10);
    pnewtask->uptr = ptrans;

    /* return T_ADVANCE so that execution continues */
    return T_ADVANCE;
}

read_done(task_ptr ptask)
{
    rw_trans_t *ptrans = ptask->uptr;

    /* Complete the load operation. */
    sou_load(ptrans, ptask->time);

    return T_FREE;
}
```

4.3 I/O Space Access

Paint includes extensions to the frontend that allow I/O space references; memory addresses that are not real memory locations (from the processor's point of view), but refer to I/O device registers that are mapped into the processor's address space. An I/O space address is any address in which the upper 4 bits of the address are set to 1. When the frontend is given such an address, it sets the processor physical and the Paint physical address equal to the virtual address. As far as the frontend is concerned, this is the only difference at the time the event is generated. The event is then passed to the backend, where it is the responsibility of the cache module to determine what to do, possibly handing the event off to another simulation module (say, a network module). At some point the I/O space device will complete operation on the event. In the case of an asynchronous write, it must free the `rw_trans_t` structure that was handed to it by the cache module. This is done with the `free_ptrans()` routine. For a load, frontend execution must be restarted by calling `sou_load()`. In both cases, the data to be read or written is contained in the `value` field of the `rw_trans_t` structure. For load instructions, the I/O space device module is responsible for placing the data in the `value` field before calling `sou_load()`. I/O space devices may operate either asynchronously or synchronously, it is up to the simulation module writer and the cache module writer.

5 Machine Dependent Interfaces

In order to run a realistic kernel and user binaries, it is necessary to support several machine dependent interfaces. System calls, traps, and interrupts are all supported in Paint. The next few sections describe these interfaces and how they are supported in both the kernel and Paint.

5.1 System Calls

To avoid special compilation of user programs, Paint must support the system call interface contained within the C Library. Under BSD and HPUX, system calls are invoked by issuing a *branch and link external (ble)* instruction to a fixed location in the virtual address space. This special location is called the *gateway page.* A production kernel will trap this branch attempt, and then transfer control to a fixed routine, while raising the privilege level and switching the virtual address context to that of the kernel. To return from the system call, the kernel will issue a *branch external (be)* instruction, which transfers control back to the user program, while lowering the privilege level and switching the virtual context back to that of the user program.

The requirements for Paint are somewhat less, since the notion of privilege level does not need to be strictly enforced. Only the transfer of control and virtual address switch needs to be implemented. Paint does this by trapping all `ble` and `be` instructions, examining the target location of the branch. For `bye`, the `event_ble` routine looks at the target address, and if it is within the first page starting at virtual address 0xC0000000, it is assumed to be a system call. Paint then transfers control to a known text address (marked by the symbol `syscall_trap`) in the kernel by locating the proper icode for that instruction, and returning it from `event_ble`. Execution continues with that instruction. When the kernel initiates a return from the system call, a similar sequence of events occurs in the `event_be` routine. The target address of the branch is checked to ensure that it is in the user's program space, after which the proper icode is located and returned. In both cases, the privilege level of the processor is changed to indicate that a switch occurred, but there is no check made when executing instructions; a user program may execute privileged instructions if it wants. So far, this has not presented any problems.

5.2 Interrupts

Processor interruptions are anomalies that occur during instruction execution which cause the flow of control to switch to a known location in the kernel. *External Interrupts* are processor interruptions which are delivered asynchronously with respect to the instruction stream. Clock interrupts and device interrupts are two types of asynchronous interrupts that are supported by Paint and the kernel. Paint provides a interface that can be used by simulation modules (say, a clock device) to generate an interrupt for the processor. For example, Paint generates a clock interrupt every 100,000 cycles by scheduling a task to call the following function:

```
vproc_interrupt{int procid, int interrupt);
```

where `procid` is the processor number, and `interrupt` is a number in the range 0 to 31 (clock interrupts are always delivered as number 31). On the PA-RISC, all external interrupts are delivered to a single interrupt handling routine, while the actual interrupt is specified using the *external interrupt request register (EIRR).* The *system priority level (SPL)* is encoded in the *external interrupt enable mask (EIEM)* register. Interrupts are masked by the processor either by clearing bits in this mask, or by turning off all interrupts via the *processor status word.* As a simplification, both Paint and the kernel currently agree to use either an *interrupts on* or *interrupts off* policy; the kernel is either masking all interrupts, or masking none of them. With so few devices generating interrupts, this is not expected to be a problem.

When an interrupt is delivered, control is transferred to the external interrupt handler routine in the kernel. This is an assembly language routine (derived from the PA-RISC version of Mach 3.0) which saves the machine state before calling the higher level interrupt handling code. When the kernel returns from the interrupt, it will restore the machine state and issue an *rfi* instruction. The interrupt handling code is entirely realistic; only a few lines of code needed to be changed for it to run under Paint.

5.3 Traps

Traps are processor interruptions that occur synchronously with respect to the instruction stream. A data page fault that results in a TLB miss would be one such example. Traps are initiated by calling the the **vproc_trap()** routine, which returns the icode of the first instruction of the proper trap handling routine in the kernel. This will be the next instruction to execute instead of the one that caused the trap. When the kernel is ready to continue the process that caused the trap, it will restore the machine state, and issue a *return from interrupt (rfi)* instruction. Typically, the kernel will restart the process at the instruction that caused the trap. To initiate a trap:

```
icode_ptr
vproc_trap(thread_ptr pthread,
           icode_ptr picode, int trapnum, rw_trans_t *ptrans);
```

The current set of supported traps are:

I_DPGFAULT	Data page fault. The requested virtual address translation is not present in the TLB.
I_LPRIVXFER	Lower privilege transfer trap. An instruction is about to be executed at a lower privilege level than the current instruction, and the PSW L-bit is set. This is used by the kernel to deliver asynchronous transfer traps (AST) in the interrupt handler.
I_DMEM_ACC	Data memory access trap. The requested virtual address translation is in the TLB, but the access permission bits are incorrect for the type of access.

6 Specifying Simulation Parameters

One of the problems we encountered was command line option explosion. Many of the simulation modules needed their own set of command line options, and it became confusing and difficult to maintain since only a single module can use an option. To address this problem, we added a parameter file that can store name/value pairs, much like the Xdefaults file in the X11 window system (although much simpler). The simulator can call the **get_parameter()** routine to get the value of a particular parameter.

The default file name for the parameter file is "./mint_params." A different filename can be specified as a command line option to the simulator (see Section 7). The format of a parameter file is very simple. Blank lines and lines beginning with a "#" are ignored. The first field on a non-blank line is the name of the parameter, and the second field is the value. For example:

```
#
# This is a sample parameter file.
#
Useful_parameter_1 0
Useful_parameter_2 1
```

A simulation module can then access the parameter values using the following function:

```
get_parameter(char *parameter_name, void *value, int type)
```

The string pointer `parameter_name` is the name used to match on in the parameter file. The match is case sensitive. `value` is a pointer to storage large enough to hold the parameter value. If no matching name is found, `value` is left unchanged. The `type` argument specifies what type of parameter value is expected. The 3 possibilities (defined in mint.h) are:

`PARAM_INT`	The value is an integer.
`PARAM_FLOAT`	The value is a float.
`PARAM_STRING`	The value is a string. The storage should be large enough to hold the largest string expected.

A simple code fragment that requests a parameter value follows:

```
{
        int         Useful_parameter = 0;

        get_parameter("Useful_parameter_1", &Useful_parameter, PARAM_INT);

        if (Useful_parameter)
            do_something();
}
```

7 Command Line Arguments

This section describes the command line options that have been changed or added to Paint.

-z	Turn on *virtual processor* mode. By default, Paint runs in SPMD mode for backwards compatibility and the virtual processor functions described in Section 2 are disabled.
-n *count*	Specify the maximum number of virtual processors when MPMD Paint is enabled. The default value is one. The kernel may create this many processors (using the `newvproc()` interface function). Attempts to create more will result in a fatal error.
-I	Turn on *Stall On Use* mode. See Section 4 for a description of SOU mode. When SOU mode is off, the backend should treat load instructions synchronously, returning T_ADVANCE only when the load is complete and execution can proceed normally. The default value is off.

-k *stack_size*	This option is ignored in MPMD Paint. Stack segments grow on demand. In SPMD mode, this option can be used only when stacks are in the shared memory space (this occurs when the `sproc` system call is used).
-m *file*	The *file* specifies a list of runtime parameters for the simulator. The default value is "mint_params." See section 6 for a description of the parameter file.
-h *heap_size*	This option applies only to the kernel heap size in MPMD Paint. The heap space for user programs grows on demand. See Section 3 for more details. The default value is 256 4K pages.

8 Simulator Support for the Kernel

There are several new support functions that are intercepted by the simulator.

8.1 `newvproc()`

Create a new simulated processor. This is functionally equivalent to `fork()` in that control returns to the parent and the child at the point following the function call. `newvproc()` returns 0 in the child, and non-zero in the parent. This function is used by the kernel startup code to duplicate itself onto as many processors as were requested when Paint was invoked (using the -n option; see section 7).

8.2 `getvproc()`

Return the current simulated processor number. The value is an integer in the range 0 to (Number_of_Nodes - 1).

8.3 `getmaxnodes()`

Return the total number of processors in the simulation. This is set using the -n option to the simulator, and is made available to the kernel and to user programs with this function.

9 Miscellaneous Notes

9.1 Building Paint

Please consult the README file in the top level directory for instructions on how to build Paint, the kernel, and the support programs.

9.2 An Example Kernel

The kernel that we run is not a production kernel, but rather a subset of the BSD 4.4 kernel (Lite-2 release) that can handle system call trapping, asynchronous interrupts, synchronous traps, signals, and provides additional functionality required by the Avalanche Scalable Computer Project. These include message passing and distributed shared memory system calls and interrupt handlers. This kernel is provided in the distribution, so please consult the README file in that directory for instructions on how to build and run the sample kernel.

9.3 Who To Contact

If you have questions or comments, please send email to avalanche@jensen.cs.utah.edu. There are several people responsible for the simulator, so this is your best chance to get a response.

The Avalanche Project requests users of this software to return to avalanche@jensen.cs.utah.edu improvements that they make and grant the Avalanche Project redistribution rights.

9.4 Credits

Thanks to Jack Veenstra (veenstra@itagain.mti.sgi.com) for developing the Mint simulator, upon which Paint is based.

References

[1] Hewlett-Packard Co. *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*, February 1994.

[2] Swanson, M., Kuramkote, R., Tateyama, T., and Stoller, L. Message Passing Support in the Avalanche Widget. Tech. Rep. UUCS-96-002, University of Utah - Computer Science Department, March 1996.

[3] Veenstra, J. Mint Tutorial and User Manual. Tech. Rep. 452, University of Rochester Computer Science Department, May 1993.