Augmint 1

1.1 Introduction

Augmint is a software package on top of which multiprocessor memory hierarchy simulators can be constructed for Intel Architecture specific platforms. The simulation platform runs native on Intel Architecture. It is currently running on Pentium platforms running the Solaris operating system and relies on the GNU gcc compiler tool chain.

1.2 Motivation

There are many simulation environments for studying memory hierarchies that exist in the research domain today. However the focus is on architectures that use RISC processors with no attention being paid to CISC architectures. Augmint was designed to address this gap by providing researchers with a simulation infrastructure that enables the modelling of systems built out of Intel Architecture processors.

1.2.1 Overview of Augmint

Augmint consists of a front end memory event generator, a simulation infrastructure which manages the scheduling of events and a collection of architectural models that represent the system under study. Those familiar with Tango Lite [1] will recognize the process of code augmentation which is explained in a little more detail below and those familiar with the MINT [2] simulator will recognize the simulation infrastructure functionality.

NOTE

However, careful attention must be paid to the details of Augmint because it does deviate from both Tango Lite and MINT in it's approach to providing a simulation environment.

The capabilities of Augmint are the following:

1. Real applications interact with the simulation environment
2. Runs on Intel Architecture platforms
3. **Recognizes x86 memory references**
4. Accounts for x86 instruction cycle times (assumes non-pipelined Pentium model)
5. Accounts for architectural models cycle times
6. Manages the scheduling of events
7. Provides a user level threads programming model
8. Provides mechanisms that allow for modelling of the movement of data to/from the application from/to the architectural models. (This concept is explained in greater detail under the section **Simulation Infrastructure**.)

Augmint consists of the following subsystems and each of these are presented below:

1. Doctor - the x86 code augmenter
2. Simulation infrastructure - the event management, task scheduling and thread switching infrastructure. This is also referred to as the back end of Augmint.
3. Architectural models - the researcher supplied components of the system to be studied
4. Applications - are written using an ANL like m4 macro package that allows for the explicit description of parallelism and shared memory as well as providing some synchronization primitives.

There are several steps in the build process when constructing the simulation executable. The application source code is first passed through the m4 macro preprocessor where all the ANL like macros are expanded. The C compiler is then run which translates the C code into x86 assembler code upon which the code augmenter is run. After code augmentation is complete, the code is assembled into an object file which is then linked with the Augmint simulation libraries. This results in an executable that contains both the application code, the architectural models under study and the Augmint simulation infrastructure. This executable, when running, consists of several user level threads that act as simulated processors and a simulation thread that acts as the event and task manager, i.e. the scheduler.

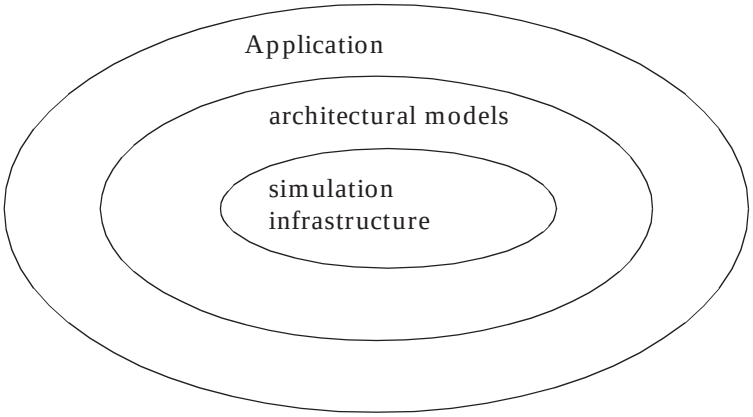

Figure 1-1: High Level Overview of Augmint Simulation

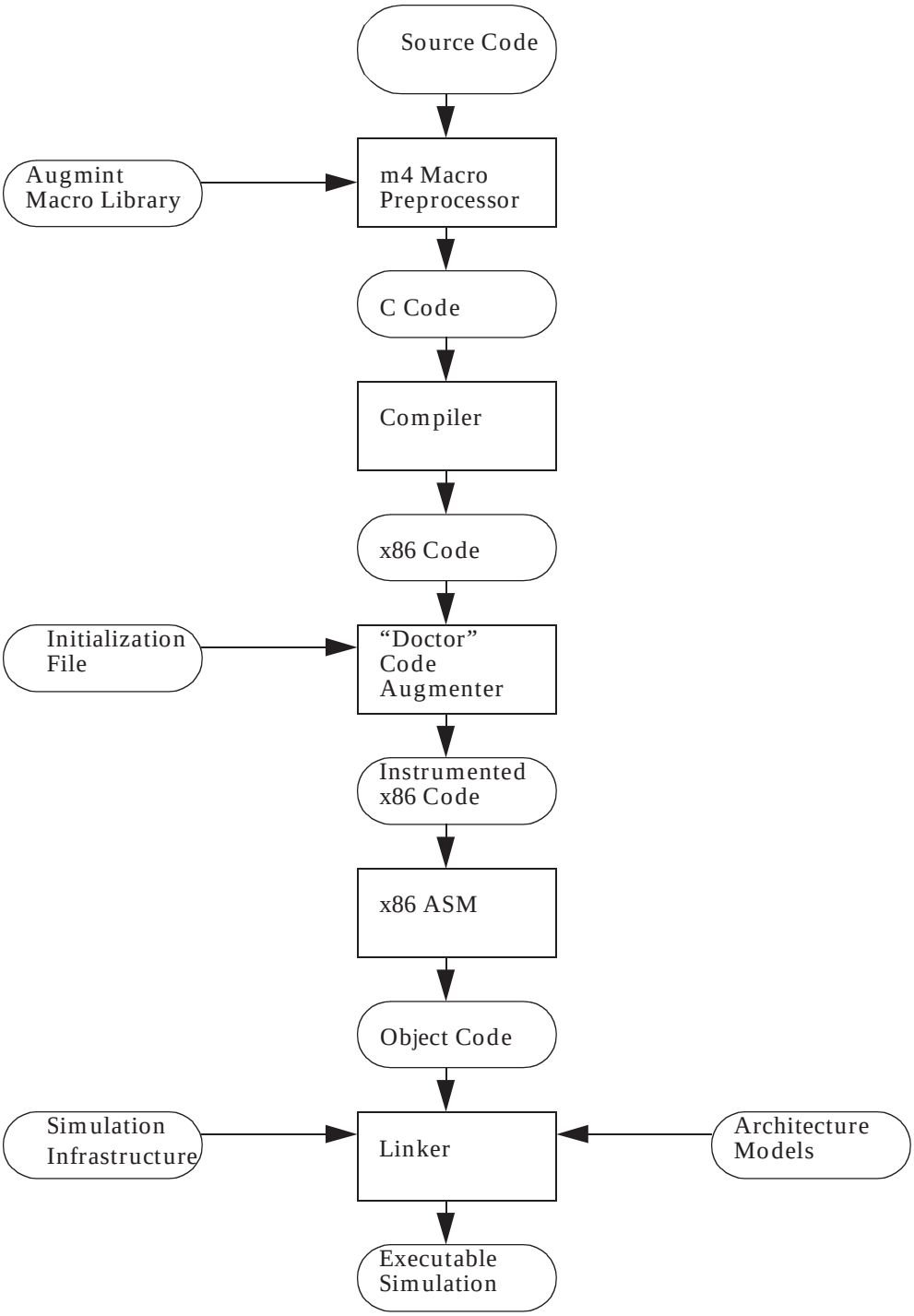

Figure 1-2: Building an Augmint Application and Simulator

1.3 Doctor

Doctor takes an 80x86 assembly source file as input and analyzes it for memory references. Each instruction that performs a memory references causes additional code to be added before or after that instruction. This code handles details like saving state and setting up the Augmint event structure as well as performing a thread switch to the simulation infrastructure thread. These mechanisms are explained in more detail later.

NOTE

Data movement is not modelled for stack references.

Doctor accounts for the time that basic blocks take to execute. Basic blocks are pieces of code that contain no “interesting” events, i.e. no events of simulation interest, which are typically blocks that contain no memory references. Doctor is a lexical analyzer that parses the x86 assembly source code and looks for instructions that cause memory reference events.

The x86 mnemonics are stored in a table in which information is kept about whether it can generate a memory reference, if so how many bytes it can address, what the category of memory reference is and how many cycles the instruction takes. *Cycle times are based on the Pentium processor*. This is the list of memory reference categories:

1. KNOWN - instruction is known to generate memory references where the destination is written and the source is read. A memory reference in a single operand instruction is assumed to be a read.
2. MEMORY_READ - a memory reference is always a read
3. MEMORY-WRITE - a memory reference is always a write
4. IMPLICIT_READ - self explanatory
5. IMPLICIT_WRITE - self explanatory
6. SPECIAL - reads / writes more than one byte
7. READ_WRITE - memory is both read and written
8. NO_REF - absolutely no memory references

1.4 The Simulation Infrastructure

The simulation infrastructure provides the context within which the architectural models exist. It provides the mechanisms for scheduling events and accounting for the passing of time. The important concepts here are: events, tasks and the task scheduler.

1.4.1 Events

Events are generated either by Doctor or through the use of the ANL like m4 macros.

Doctor recognizes memory access events of interest in the application assembly code and through the process of code augmentation notifies the back end that an interesting event is about to occur. The augmented code sets up the event structure and performs a thread switch to the simulation thread which decodes the event and schedules a task to handle it.

Each event is represented by a structure that contains the process id of the thread that generated the event, the time at which the event occurred, and the type of event. The event structure also includes some fields that are used only by certain events.

NOTE

Augmint deviates from MINT in that it assumes that the parallel programming model is based on threads instead of processes. A multi-process parallel application will work under Augmint as long as the processes do not have their own regions of global memory. The implication here is that the event fields ***vaddr*** *and* ***paddr*** *are identical.*

Some operations generate an event after performing the operation. These operations include forking a new thread, acquiring a lock, acquiring a barrier, acquiring a semaphore, unblocking a process, and terminating. These operations require that the operation be performed before the event is generated. For reasons of atomicity and correct ordering of events, the **E_LOCK_ACQUIRE** event (and the corresponding events for acquiring barriers and semaphores) must not be generated before the lock is acquired, since otherwise some other process may acquire the lock instead.

The event structure is defined in **event.h**:

```
typedef struct event {

            mint_time_t time;

            mint_time_ptr cpu_time;

            mint_time_t duration;

            struct event *next;

            int pid;

            long type;

            int  size;

            long utype;

            long vaddr;

            long paddr;
```

```
        long value[MAX_BLOCK];

        struct event *pevent;

        void *sptr;

        char *fname;

        long arg1;

        long arg2;

        long arg3;

        long arg4;

        long rval;

} event_t, *event_ptr;
```

The meaning of each field is given below.

time - The time (in processor cycles) at which this event was issued. To obtain the current time, the back end should use the time field in the task structure.

cpu_time - A pointer to the accumulated cpu time (in processor cycles) for this thread. The cpu time does not include the cpu time of children. The back end may use this pointer to change the cpu time of a process, but in most cases this is not necessary.

duration - If this is an unblock event, then duration contains the time that this thread spent waiting.

next - A pointer for linking together a list of events.

pid - The process id of the thread causing the event. The first thread has a pid of zero.

type - The type of event. (See **event.h** for macro definitions.)

vaddr and paddr - The virtual address or the memory referenced.

value - The value to read or write at paddr. If protocol verification is being used (data movement) this value is filled in by the back end on a read event, and this value is used by the back end for writing on a write event.

pevent - A pointer to the event structure for another thread. On a fork, this field is set to point to the event structure for the child. On an **E_UNBLOCK** event, this points to the event structure for the thread that did the unblocking. A thread may unblock another for the following reasons: (1) when a thread terminates it will unblock its parent, if necessary (so that it can **wait()** on its child), (2) when a thread executes a V() operation on a semaphore, it will unblock another thread that previously executed a P() operation for

that semaphore, (3) when a thread releases a lock, it will unblock a thread if there is one waiting for the lock, and (4) when a thread reaches a barrier and it is the last thread to arrive, it unblocks all the other threads.

sptr - A pointer field that is used by the task scheduler. This field must not be modified.

fname - If the event type is **E_FUNCTION** then this field contains a pointer to the function name that was intercepted by Augmint.

Types of events

The types of events supported by Augmint are listed in **event.h**. This header file includes macro definitions for each of the possible types.

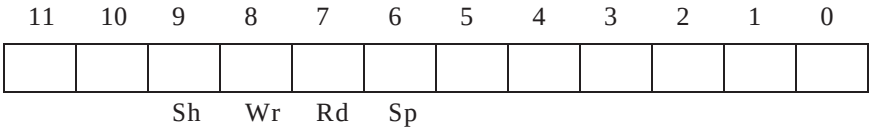

Figure 1-3: Bits fields that encode event type

E_SPECIAL - If set, then the lower order bits are used to encode special events. These events are listed below in the description of special events.

E_READ - This is a read operation.

E_WRITE - This is a write operation.

E_SHARED - This is a shared read or shared write operation.

Each event type is the bitwise OR of some of the above bit fields. Each of the possible event types has a corresponding macro definition given below. For most purposes, the back end need not examine the event type since each different event causes a separate function to be called.

Special events indicate a change in thread status or number of threads:

E_FORK - This thread forked. The **pevent** field contains a pointer to the child thread.

E_LOCK_ATTEMPT - This thread is about to attempt a lock operation. The **arg1** field contains the address of the lock.

E_LOCK_ACQUIRE - This thread has successfully acquired a lock. The **arg1** field contains the address of the lock.

E_LOCK_RELEASE - This thread is about to release a lock. The **arg1** field contains the address of the lock.

E_BARRIER_ATTEMPT - This thread is about to attempt a barrier operation. The **arg1** field contains the address of the barrier.

E_BARRIER_ACQUIRE - This thread has successfully acquired a barrier, which means that it was the last thread to reach the barrier. The **arg1** field contains the address of the barrier.

E_PSEMA_ATTEMPT - This thread is about to perform a P() operation on a semaphore. This may cause the thread to block if the semaphore is not available. The **paddr** field contains the address of the semaphore.

E_PSEMA_ACQUIRE - This thread has successfully acquired a semaphore without blocking.

E_VSEMA - This thread is about to perform a V() operation on a semaphore. This may unblock another thread if one is waiting. The **arg1** field contains the address of the semaphore.

Tasks

A realistic simulation of a parallel processor must allow multiple concurrent events to occur. A memory read by processor 1, for example, may execute simultaneously with a memory write by processor 2. Each of these operations may take a variable amount of time, and may require other operations to be performed, such as reading a cache block over the bus, writing a dirty block back to memory, or invalidating other processors caches.

To ensure that all operations are simulated in chronological order, Augmint provides support for creating and scheduling arbitrary operations. Each independently schedulable operation is called a **task**. Each task has associated with it a time value, a priority, and a function pointer. Augmint executes each task at its specified time in priority order. Tasks are executed by calling the function through the function pointer. The return value from this function controls thread execution.

Every event generated by a thread blocks that thread and creates a new task. The function pointer for the task points to the architecture simulator function corresponding to the type of event. The task also contains a pointer to the event structure and the process id.

Return values for tasks are defined in **task.h** and are listed below:

T_ADVANCE - The processor associated with this task may continue executing instructions. A common mistake is to return **T_ADVANCE** from a task when the processor is not blocked. The debugging version of Augmint detects this type of error. (Use ``make debug'' or ``make versions'' when building Augmint to get the debugging version.)

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. If the processor associated with this task is blocked, then it remains blocked.

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. This is normally used only by routines internal to Augmint.

T_CONTINUE - This return value specifies that the next task on the task stack should be scheduled now. If the task stack is empty, then this is equivalent to (but slightly less efficient than) using **T_ADVANCE**. Tasks are put on the task stack by calling **stack_task()** or **pri_stack_task()**. A task stack is associated with a task, not with a thread, so multiple independent stacks can be active for the same thread simultaneously. The intended use of this is to support modularization. For example, a bus or network module can be written with a single entry point and such that it always returns **T_CONTINUE**. The module can then be called from several different points in the code and after the module completes the simulation will continue at the function pushed on the stack before the call to that module. (See the call **pri_stack_task()** in the **Augmint Interfaces** section.)

Augmint provides mechanisms that allow the researcher to create and schedule tasks and they are described in the section **Augmint Interfaces**.

The task structure is defined in **task.h**:

```
typedef struct task {
            struct task *next;
            struct task *prev;
            mint_time_t time;
            int priority;
            int pid;
            int (*ufunc)(struct task *);
            struct task *tstack;
            struct event *pevent;
            int ival1;
            int ival2;
            int ival3;
            int ival4;
            int ival5;
            int ival6;
            int ival7;
            int ival8
            int ival9;
            int ival10;
```

```
                int ival11;

                int ival12;

                double dval1;

                double dval2;

                mint_time_t tval1;

                mint_time_t tval2;

                mint_time_t tval3;

                void *uptr;

                void *uptr2;

                void *uptr3;

                void *uptr4;

                int (*ufunc2)(struct task *);

                struct task *unext;

                struct task *uprev;

    } task_t, *task_ptr;
```

next - A forward link for task queues.

prev - A backward link for task queues.

time - The time at which this task is scheduled.

priority - The priority level. Among tasks scheduled at the same time, higher numbered priorities are executed first. The default priority is zero. Negative numbers may be used to schedule low priority tasks.

pid - The process id.

ufunc - The function to call when this task is scheduled.

tstack - The task stack. This is initially empty. Tasks may be pushed onto the stack by calling **stack_task()** or **pri_stack_task()**. The task at the top of the stack is scheduled when the current task returns **T_CONTINUE**.

pevent - A pointer to the event associated with this task.

ival1 - An arbitrary integer value. Not used by Augmint.

ival2 - An arbitrary integer value. Not used by Augmint.

ival3 - An arbitrary integer value. Not used by Augmint.

ival4 - An arbitrary integer value. Not used by Augmint.

ival5 - An arbitrary integer value. Not used by Augmint.

ival6 - An arbitrary integer value. Not used by Augmint.

ival7 - An arbitrary integer value. Not used by Augmint.

ival8 - An arbitrary integer value. Not used by Augmint.

ival9 - An arbitrary integer value. Not used by Augmint.

ival10 - An arbitrary integer value. Not used by Augmint.

ival11 - An arbitrary integer value. Not used by Augmint.

ival12 - An arbitrary integer value. Not used by Augmint.

dval1 - An arbitrary double value. Not used by Augmint.

dval2 - An arbitrary double value. Not used by Augmint.

tval1 - An arbitrary time value. Not used by Augmint.

tval2 - An arbitrary time value. Not used by Augmint.

tval3 - An arbitrary time value. Not used by Augmint.

uptr - An arbitrary pointer. Not used by Augmint.

uptr2 - An arbitrary pointer. Not used by Augmint.

uptr3 - An arbitrary pointer. Not used by Augmint.

uptr4 - An arbitrary pointer. Not used by Augmint.

ufunc2 - A function pointer. Not used by Augmint.

unext - A forward link for queues. Not used by Augmint.

uprev - A backward link for queues. Not used by Augmint.

The fields that are not used by Augmint may be used by the researcher to hold state between calls to back-end functions. Since these fields might not be needed by the back end, they are not initialized to zero and may contain garbage values. The back end is responsible for initializing the fields that it uses.

Task Scheduling

Tasks are added to a structure known as the time wheel, which sorts them out by time and then within the same time by priority. Events are generated which cause tasks to be created and added to the time wheel. Tasks will often times create more tasks and it is up to the researcher to determine when these tasks should be executed. The interfaces for creating and scheduling tasks are listed in the section **Augmint Interfaces**.

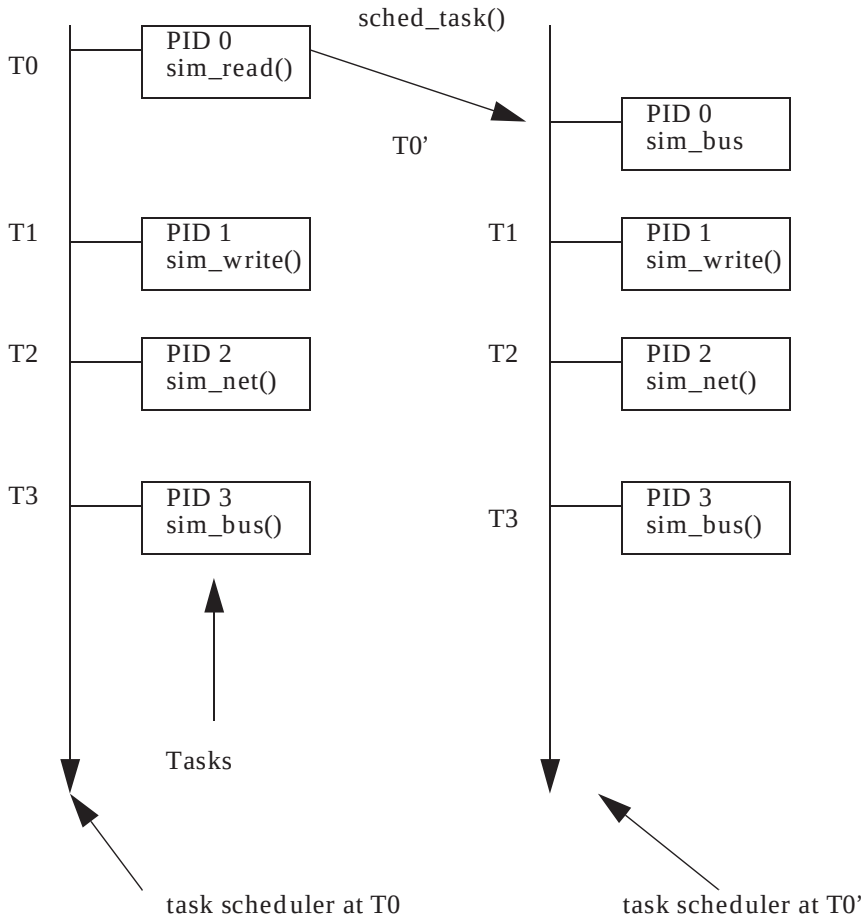

Figure 1-4: The task scheduler

1.5 Augmint Interfaces

1.5.1 Functions called by Augmint:

These functions are called by Augmint and may be defined by the back end. All of these functions are optional. If one is not supplied, an empty stub function will be linked in from the Augmint library.

int sim_init(int argc, char **argv)

Augmint calls **sim_init()** once at the beginning of execution. This function is typically used to initialize data used by the back end. Augmint passes its own **argc** and **argv** to **sim_init()** so that the entire command line is available. This includes the arguments to Augmint, which appear first, followed by the double hyphen separator (`` --''), followed by the arguments to the back end.

The simulation is started by invoking the application:

application [Augmint options] [-- back-end options]

Augmint reads its command line options using the **getopt()** library function. (See section **options** for a description of the command line options for Augmint.) Since **getopt()** skips over the double hyphen separator (``--'') the next call to **getopt()** will return the first back-end option, if any. This makes it convenient for the back end, since the back end options can be parsed using **getopt()** as if they were the only options on the command line. If the back end is composed of separate independent modules each of which has its own set of options, the convention of using (``--'') to separate one module's options from the next makes it easy to parse each module's options independently.

So a typical **sim_init()** looks like:

```
sim_init(int argc, char **argv) {
            extern int optind;     int c;
            while ((c = getopt(argc, argv, "a:b:o:")) != -1)
            switch (c) {
                        /*... do something with the options ...*
                        */
            }
            return optind;
}
```

void sim_done(double elapsed, double cpu_time)

Augmint calls **sim_done()** once at the end of execution, after all threads from the application program have terminated. This function is typically used to print statistics collected during the run. The first argument is the simulated execution time of the application in processor cycles, and the second argument is the total accumulated cpu time of all the simulated threads.

void sim_usage(int argc, char **argv)

This is called if there was an error parsing the command line. Augmint first prints its own usage message before calling **sim_usage()**.

int sim_read(task_ptr ptask)

int sim_write(task_ptr ptask)}

int sim_user(task_ptr ptask)

int sim_lock_attempt(task_ptr ptask)

int sim_lock_acquire(task_ptr ptask)

int sim_lock_release(task_ptr ptask)

int sim_barrier_attempt(task_ptr ptask)

int sim_barrier_acquire(task_ptr ptask)

int sim_psema_attempt(task_ptr ptask)

int sim_psema_acquire(task_ptr ptask)

int sim_vsema(task_ptr ptask)

int sim_exit(task_ptr ptask)

int sim_function(task_ptr ptask)

int sim_terminate(task_ptr ptask)

int sim_kill(task_ptr ptask)

int sim_alloc(task_ptr ptask)

int sim_shalloc(task_ptr ptask)

int sim_lockalloc(task_ptr ptask)

sim_read() is called on every read event, passing it a pointer to a task. Similarly, **sim_write()** is called on every write event, and **sim_user()** is called on every user-defined event. User defined events are caused by using the macro **GEN_USER_EVENT**. This is a convenient mechanism whereby the application can communicate with the simulation infrastructure.

sim_lock_attempt() - is called when a thread attempts to acquire a spinlock. This occurs when the application uses the m4 macro **LOCK()**. Augmint calls **sim_lock_acquire()** when a process acquires a spinlock, and **sim_lock_release()** when a process releases a spinlock. Locks are released when the application calls the macro **UNLOCK()** . Spinlocks are costly to simulate because they can cause an event every cycle. To reduce the overhead of simulating spinlocks, Augmint blocks a process if it fails to acquire a spinlock on the first attempt and unblocks it only when the lock becomes free. This results in only one call to **sim_lock_attempt()** instead of one every cycle, and coincides with the behavior that would occur if queue-based locks were used. This optimization can be turned off with the [**-S**] command line option.

sim_barrier_attempt() is called when a thread reaches a barrier. Augmint calls **sim_barrier_acquire()** when a thread continues past the barrier. Barrier events are caused by the application using the macro **BARRIER()**.

sim_psema_attempt() - when a thread uses the macro **PSEMA()** to attempt a P() operation on a semaphore, and **sim_psema_acquire()** when the P() operation succeeds. Augmint calls **sim_vsema()** when a thread performs a V() operation on a semaphore. If several threads are waiting on a semaphore when a V() operation is performed, then exactly one waiting thread is unblocked.

To force synchronization variables to pay for the cost of reading and writing memory, the library versions of the synchronization routines may call either **sim_read()** or **sim_write()**. For example, the code for the library function **sim_lock_acquire()** is:

```
sim_lock_acquire(task_ptr ptask) {

            return sim_write(ptask);

}
```

sim_lock_attempt() is the only library function that calls **sim_read()**. The library functions that call **sim_write()** are the following: **sim_barrier_attempt()**, **sim_lock_acquire()**, **sim_lock_release()**, **sim_psema_attempt()**, and **sim_vsema()**. If a behavior different from the default is desired, the user should define these functions in the back end. With the exception of three of the above functions, Augmint calls these functions **before** the operation is performed. This allows the back end to block the thread that generated the event and exercise control over access to memory and synchronization variables. The exceptions are: **sim_lock_acquire()**, **sim_barrier_acquire()**, and **sim_psema_acquire()**. The three exceptions are called **after** the process has successfully completed the operation..

Augmint calls **sim_terminate()** as the last back-end function call before a thread is terminated. If there are still tasks scheduled for this thread, a warning message is printed.

sim_alloc() - is called whenver the **MALLOC()** macros is used in the application program. **sim_shalloc()** is called when a thread allocates shared memory using the **G_MALLOC()** macro. The event structure includes fields that are useful for constructing a memory allocation map:

1. **pevent->arg1** - This is the size of the memory requested by the application.

2. **pevent->rval** - This is the base of the memory region returned by the call to **malloc()**, **calloc()**, etc.

void sim_fork(task_ptr ptask)

void sim_block(task_ptr ptask)

void sim_unblock(task_ptr ptask)

void sim_checkpoint(task_ptr ptask)

void sim_message(mint_time_t curtime, char *arg)

sim_fork() is called when a thread forks a child. The **pevent** pointer in the task structure points to the event for the parent thread. The pointer to the child's event is stored in the **pevent** field of the parent's event structure.

sim_block() - is called when a process blocks, and **sim_unblock()** when a process becomes unblocked. A process may block for any of the following reasons:

1. the thread exits before all of its children have finished

2. the thread calls **wait()** and there are child threads but none of those that have not already been waited on have finished

3. the thread performs a P() operation on a semaphore and the semaphore is not available

4. the thread reaches a barrier and it is not the last thread to reach the barrier

5. the thread attempts to acquire a lock that is held by some other thread (unless spinning is enabled with the [**-S**] option).

The task pointer passed to **sim_unblock()** is for the *unblocked* thread. The event pointer for the unblocking thread can be retrieved through **ptask->pevent->pevent**. The number of cycles that the unblocked thread remained blocked is stored in the **duration** field of the event structure (**ptask->pevent->duration)**. A separate call to **sim_unblock()** is made for each unblocked thread.

Augmint schedules a checkpoint task to execute at fixed intervals. This task does not really do any checkpointing in the sense of saving the simulation state to be restarted in case of a system crash, but it does provide a convenient hook for the back end to perform some operation at regular intervals. The default checkpoint interval is ten million cycles. This can be changed to any value by using the command line option **-C interval**, where **interval** is the number of cycles between checkpoints. Setting the interval to zero turns off

checkpointing. Each time the checkpoint task runs, it calls the function **sim_checkpoint()**. If the back end does not define this function, the library version is linked in which simply prints a message containing the time.

1.5.2 Functions defined by Augmint

These functions are supplied by Augmint and are callable from the architectural models.

task_ptr sched_task(task_ptr ptask, PFTASK ufunc, mint_time_t time)

This function creates a new task and schedules it at the given **time** with a **priority** of zero. A pointer to the current task is passed in as the first argument. This is required because every task is associated with some thread and event. This association is passed on to the new task. When the new task is executed, the user-defined function **ufunc** is called with a pointer to the new task structure. **sched_task()** returns a pointer to the new task. This pointer may be used to fill in the user-defined fields.

task_ptr pri_sched_task(task_ptr ptask, PFTASK ufunc, mint_time_t time, int priority)

This function is similar to **sched_task()** with the addition of a **priority** level argument. The priority is used to order tasks that are scheduled to run at the same time. Higher numbered priorities execute first. Negative values may also be given. Calls to **sched_task()** may be mixed with calls to **pri_sched_task()**. Tasks scheduled with **sched_task()** have a priority of zero. A pointer to the new task is returned.

task_ptr stack_task(task_ptr ptask, PFTASK ufunc)

In complex simulations it is sometimes convenient to be able to specify that a task should execute when a sequence of activities completes. The completion time and the number of intermediate tasks that need to be scheduled may not be known. The **stack_task()** function provides a way to specify that a task should be scheduled whenever another task completes. This function creates a new task to execute the function pointed to by **ufun**c. The new task is not scheduled but is pushed on the task stack for **ptask**. Whenever a task returns **T_CONTINUE**, the stack for that task is popped and the task that was at the top of the stack is scheduled. The stack for a task is passed along to newly created tasks by calls to **sched_task()** or **pri_sched_task()**.

task_ptr pri_stack_task(task_ptr, PFTASK ufunc, int priority)

This function is similar to **stack_task()** with the addition of the priority argument. A new task is created and pushed onto the task stack of ptask. When the new task is popped from the stack (because some task returned **T_CONTINUE**), the new task is scheduled at that time with the specified **priority**.

task_ptr newtask()

This creates a new task without initializing any fields. In particular, the **pevent**, **pid**, and **priority** fields may have garbage values and should be initialized by the back end, if needed. The new task can be scheduled with a call to **sched_task()**. Before scheduling this **task**, the priority field should be initialized. In addition, if this task is not associated with

any thread id, then the **pid** field should be set to an invalid value, such as -1. This task should not return **T_ADVANCE** unless the **pid** field is set to a valid process id. A pointer to the new task is returned.

void free_task(task_ptr ptask)

This frees the task pointed to by **ptask.** This function does not remove the task from the queue of scheduled tasks. To do that, use **cancel_task()**.

task_ptr schedule_task(task_ptr pnew)

This schedules the task pointed to by **pnew**.

void cancel_task(task_ptr ptask)

This removes the task pointed to by **ptask** from the task queue and frees it. *If this routine is called with a pointer to a task that has not been scheduled, bad things will happen*.

task_ptr unblock_proc(int pid, mint_time_t time)

A new task is created and scheduled at the given time. This task is associated with processor **pid** and, when executed, will unblock that processor. That processor will then execute instructions until it generates an event.

void mem_ref(long *value, long paddr, int type)

This function handles all the possible read and write operations, **value** is a pointer to the value to read or write, **paddr** is the memory address to read or write, and **type** is the type of event.

1.5.3 Architectural Models

This section contains simple architectural models that demonstrate how to use Augmint.

A simple example

This example model simply counts the number of reads and writes for private and shared memory references. The code for these modules is in the file ``example1.c'' in the examples directory.

The modules need to include the header file **mint.h** in order to get structure definitions and function prototypes.

```
#include “mint..h”
```

Since we want to count both private and shared memory references for each read and write, we define a structure so that we do not have to keep inventing different variable names for the same concept.

```
/* define a type for memory reference stats: shared and private */

typedef struct memref_t {

            int prv;

            int shr;

} memref_t;
```

We also define a structure to store the global counts. This groups all the counters in one place so it is easy to see what we are counting and makes it easy to add new fields later.

```
struct global_stats {

            memref_t nreads;

            memref_t nwrites;

} Data;
```

Next, we need to define the function **sim_read()** which is called on every read, and **sim_write()** which is called on every write. The type field is extracted from the event structure and the **E_SHARED** bit is checked to determine if this is a shared reference.

```
sim_read(task_ptr ptask) {

            int type;

            type = ptask->pevent->type;

            /* count shared and private reads */

            if (type & E_SHARED)

                        Data.nreads.shr++;

            else

                        Data.nreads.prv++;

            return T_ADVANCE;

}
```

The sim_write() function is similar to **sim_read()**:

```
sim_write(task_ptr ptask) {

            int type;

            type = ptask->pevent->type;
```

```
                /* count shared and private writes */
                if (type & E_SHARED)
                        Data.nwrites.shr++;
                else
                        Data.nwrites.prv++;
                return T_ADVANCE;
    }
```

To print the results at the end of the simulation, we need to define the function **sim_done()**. The arguments passed in are the simulated elapsed time and the total simulated cpu time used by all processes.

```
    void sim_done(double elapsed, double cpu_time) {
                printf("Private: %d reads, %d writes\n",
                        Data.nreads.prv, Data.nwrites.prv);
                printf("Shared:  %d reads, %d writes\n",
                        Data.nreads.shr, Data.nwrites.shr);
    }
```

That's the entire program! Note that the Augmint infrastructure allows the researcher to focus only on the details of interest.

1.6 Building a Parallel Application

Simulated processors under Augmint are represented by threads. There is one address space per program as a result. All data is shared except for the thread's stack space. Applications that rely on Unix process semantics for parallelism will have to be modified in order to run properly under Augmint.

1.6.1 Augmint m4 Macro Package

Those familiar with the Argonne National Labs m4 parallel programming environment will recognize the following m4 macros. The complete m4 macro package as it comes from ANL is not supported. Many macros are null macros and are included only so that the SPLASH II benchmarks can be built and run under Augmint.

MAIN_ENV - This defines the global symbol, variables and functions that make up the Augmint environment. This macro needs to appear at the beginning of the main source file for the application.

EXTERN_ENV - Declares the symbols, variables and functions that define the Augmint environment. In contrast to **MAIN_ENV**, this macro declares variables as extern. This needs to appear in every source file except the main source file for the application.

MAIN_INITENV(share-mem) - Under Augmint this is a null macro.

CLOCK(variable) - Set the specified variable to the current simulation time.

MAIN_END - This is a null macro also, which is included for compatibility with SPLASH II applications.

The following macros are used to express parallelism and established shared memory regions.

CREATE(function_name) - Creates and initializes a new Augmint user thread which calls the function specified in function_name. The called function should have no arguments.

G_MALLOC(amount) and **G_MALLOC_F**(amount) - Create shared memory regions. Each returns a pointer to a shared region of size amount bytes. The **G_MALLOC** macro is terminated by a semicolon, so it cannot be used in a C expression; otherwise the two are identical.

G_FREE(pointer) - This releases a block of shared memory that was allocated with **G_MALLOC()**.

Synchronization Macros

LOCKDEC(name) - Declares *name* to be a lock variable. Locks must be allocated out of shared memory.

LOCKINIT(name) - Initializes the lock variable *name*. Locks are initially in the unlocked state.

LOCK(name) - acquires the lock variable with the specified *name*. Locks can only be acquired by one thread at a time and if the desired lock is busy, the thread blocks until it is released.

UNLOCK(name) - Releases the lock variable *name*.

ALOCKDEC(name,number) - Declares an array of number lock variables with the specified name. Lock arrays must be declared in shared memory. *These locks are packed, which may result in false sharing!*

ALOCKINIT(name,number) - Initializes an array of *number* lock variables with the specified *name*. All locks are set to unlocked.

ALOCK(name,i) - Locks the i^{th} lock in a lock array specified by *name*.

AUNLOCK(name,i) - Releases the i^{th} lock in a lock array specified by *name*.

BARDEC(name) - Declares a barrier with the specified *name*. Barriers must be declared in shared memory.

BARINIT(name) - Initializes a barrier with the given *name*.

BARRIER(name,number) - Causes the thread to enter the specified barrier *name*. Each thread entering a barrier must wait until the specified *number* of threads have reached the barrier. Once all threads have reached the barrier, they will be released.

WAIT_FOR_END(num) - This macro causes the thread to wait for the completion of *num* other threads before proceeding.

Simulation Control Macros

AUG_ON and **AUG_OFF** - These are used to statically mark sections of the code that are not to be augmented. For example, this allows data to be initialized at full speed, before simulating references of interest.

1.7 Inside an Augmint Simulation

The mechanisms within Augmint have been exposed in the previous sections. This section provides a brief overview of how these all fit together when running a simulation model.

The simulation starts by initializing the Augmint environment:

1. The command line arguments are parsed by the architecture simulator and application.

2. **sim_init** is then called which can be just a stub or redefined by the architecture simulator writer.

3. Thread structures are initialized.

4. The address space is initialized.

5. The main application thread is initialized and set to point to the function **main_envelope** which calls the application.

6. The time wheel is initialized.

7. An empty task is scheduled for the main application thread on the time wheel. The purpose of this task is to invoke a thread switch to the main application thread to start running **main_envelope**.

After initialization, the simulation runs in a scheduling loop that repeatedly extracts tasks from the time wheel and executes them. Some of these tasks will schedule new tasks, create new threads, terminate a thread, or return **T_ADVANCE** (or **T_CONTINUE**) thus invoking a thread switch to an application thread.

The first task extracted from the time wheel is the empty task used to invoke a thread switch to the main application thread which executes **main_envelope**. The function **main_envelope** starts by calling the function **appl_main(argc, argv)**. The rest of **main_envelope** serves to terminate the main application thread by generating events and thread switching back to the Augmint simulator thread. The main application thread is checked to see if all tasks have been executed, that all children threads have terminated before it itself is finally terminated. In the case of existence of tasks or children threads, the main thread gets blocked until these conditions are satisfied, and only then does it terminate. Once all application threads have terminated the simulation thread exits the thread scheduling loop and calls the **sim_done()** function.

A task that returns **T_ADVANCE** (or **T_CONTINUE** with no stacked tasks) causes a thread switch to the application thread that it was running on its behalf. Otherwise, the application thread remains blocked, and Augmint extracts another task from the time wheel which might or might not belong to the same application thread.

1.7.1 Generating an event

While running an application thread, the variable **pcur_thread** points to the thread structure of that thread. An application thread generates an Augmint event by setting some fields of the event structure pointed to by **pcur_thread->pevent**. The most important of these fields is the field **type** which determines the event type. After that the application thread switches back to the Augmint simulation thread. which is pointed to by the variable **psim_thread**. This process of setting the fields of **pcur_thread->pevent** and thread switching to the Augmint simulation thread is usually transparent to the application writer by using the ANL like macro package or through the process of augmentation.

1.8 Conclusion

Augmint enables researchers to concentrate soley on the pieces of the system that they wish to study by providing a rich simulation infrastructure. This simulation environment is unique in that it also provides researchers with the tools to model systems built around the Intel Architecture processors line.

1.9 Appendix

1.9.1 Debugging Architectural Models

The compiler tool chain used for building and debugging Augmint based simulations is the GNU tool chain. Here is a list of helpful practices for debugging:

1. Whenever applicable use **assert(3)** to catch cases that should not happen.

2. Include simulator command line options and functionality to support debugging.

3. Support a function to trace important information. For example: task name, time, initiating processor id, local node (the node where the task is taking place), address or memory line id, directory and/or cache states for memory line on the local node, the home node of the memory line, and value at memory.

1.9.2 Common bug symptoms:

1. Assertion failure.

2. Simulated system deadlock.

3. Simulation deadlock.

4. Stray pointers.

Core Dump

Assuming that assertions are used wherever applicable, core dumps are more likely to be caused by simple easily identified programming mistakes.

Assertion failure

The first step would be to run the simulation under a debugger and determine the variable values at the time of the assertion failure. In most cases the problem can be identified at that point. If not, then you need to collect traces for several thousand cycles before the assertion failure. If the trace indicates that a problem happened earlier, then either collect earlier traces or if applicable add assertions to the code to catch these cases. Traces should help identify the cause of the problem.

Simulated system deadlock

In this case Augmint discovers that there are no more tasks although at least one thread is not done yet. To solve this problem:

1. Look at the Threads[] structure in a debugger and determine which threads are not done yet.

2. Collect traces and identify the last tasks of these threads. In most cases the mistake would be that a task returns **T_FREE** while it should have returned **T_ADVANCE**.

3. Otherwise there might be a flaw in the simulation model. Identifying the last several tasks of deadlocked threads helps identifying the problem.

Simulation deadlock

The most probable cause for this problem is a protocol bug. To find the bug generate a trace until a point long enough after you believe the deadlock state happened. Search for the last **sim_read** or **sim_write**, find which memory line it was for. Do a grep for that memory line and analyze backwards looking for any inconsistent states.

Another possibility is a programming bug that occurs at a certain simulated time which you can identify by analyzing the trace. Inside a debugger you can breakpoint at a certain time by defining breakpoints with conditions. However this tends to slow the simulation substantially. Another alternative provided by Augmint is to:

1. Define a breakpoint at the function debug_task (breakpoint 1).

2. Define another breakpoint at line dbg_task.c:17 (breakpoint 2).

3. Run the simulation. It will stop at breakpoint 1.

4. Disable breakpoint 1.

5. Assign the time value you want to break at to the variable Debug_time. For example in gdb: “print Debug_time = 377898”.

6. Continue the simulation.

7. The simulation will stop at dbg_task.c:17 only when ptask->time is at least 377898 without significant loss in speed.

8. From this point run the program in single steps till you discover the problem.

Stray Pointers:

Stray pointers (or indices) can cause all kinds of bugs and they are extremely hard to find. An indication that a problem is caused by a stray pointer is if you find that a variable changes its value between two points in a trace while all the tasks between these points should not modify that variable. A relatively easy approach to looking for these stray pointer bugs is as follows:

1. By repeated use of time-based breakpoints, try to identify a small time period (say 100 cycles) where the variable changes.

2. Perform steps 1-7 as mentioned above in order to stop the simulation at a time where you believe that the value of the variable is consistent and where its value will change shortly after to an unexpected value.

3. From this point print the variable's address "print &var". The response of the debugger would be for example "$15 = (var_t *)0xbadf00d"

4. Define a "display" for that pointer (e.g. display *$15, or display *(var_t *)0xbadf00d). We use a display for the pointer because the variable might be local.

5. Define another display "ptask->time".

6. Repeatedly continue the simulation, every time the simulation stops the debugger displays "ptask->time" and "var". At some point the value of "var" changes. Now you know the time it happened (say 222889).

7. Perform steps 1-7 in the previous section for time 222889. The simulation stops at dbg_task.c:17 at time 222889.

8. Define a display for "var" and single step from this point.

9. At some point (soon) you will notice that the value of "var" has changed and by simple analysis you would know when and why this happened.

Basic Block Checking

When testing a simulator, knowing which basic blocks of the simulator has been executed during these tests, helps evaluate the reliability of these tests. One way of doing this is to:

1. Compile the simulator code files of interest with gcc option -a.

2. Running a simulation will result in a file "bb.out" that contains information about how many times each basic block in the files of interest have been executed.

3. If the bb.out file is not in the same directories as the files of interest, copy it to these directories.

4. Invoke emacs and visit the files of interest.

5. Get into the compile environment of emacs (type M-x compile).

6. At the command prompt type "gawk -f bbp.awk bb.out", where bbp.awk is an awk program that generates compilation output in the emacs compilation environment.

7. By clicking on a line in the compilation output, emacs displays the beginning of the corresponding basic block.

An example of such an awk program (bbp.awk) is:

BEGIN { printed = 0; fname = "junk" }

$1 == "Block"

```
{
    if ($2 == "#") {
                func_name = $10;
                line_num  = $12;
                prog_name = $14;
                exec_time = $5;
    }
    else {
                func_name = $9;
                line_num  = $11;
                prog_name = $13;
                exec_time = $4;
    }
    if (fname != func_name) {
                printed = 0;
                line = 0;
    }
    if ((exec_time != 0) && (line != line_num)) {
                if (printed == 0) {
                            printf ("\n%s:\n", func_name);
                            printed = 1;
                            fname = func_name;
                }
                line = line_num;
                printf ("%s:%s: %s\n", prog_name, line_num, exec_time);
    }
    }
```

```
}

END   { printf "\n" }
```

1.10 Bibliography

[1] Stephen Alan Herrod. Tango Lite: A Multiprocessor Simulation Environment. Technical Report Stanford University, November 1993.

[2] Jack E. Veenstra, Robert Fowler. MINT Tutorial and User Manual. Technical Report 452, University of Rochester, June 1993.