DESim: A Framework and Methodology for Detailed Computer Architectural Simulation

Christopher E. Giles[†], Christina L. Peterson[‡], Mark A. Heinrich[‡]
[†]Department of Electrical and Computer Engineering
[‡]Department of Computer Science
University of Central Florida, FL, USA
{christopher.e.giles, clp8199}@knights.ucf.edu, heinrich@cs.ucf.edu

***Abstract*— We introduce DESim; a framework and methodology for detailed computer architectural simulation. DESim extends an existing, but little discussed computer architectural simulation methodology and incorporates enhancements that allow the framework to run in modern 32bit and 64bit Linux environments. As a framework, DESim provides the tools necessary to create fine-grained cycle-level computer architectural simulations. Researchers can utilize DESim's built-in functions to rapidly develop both small-scale and large-scale functional and power computer architectural simulation models. As a methodology, DESim provides the means for researchers to model complete computational systems as a collection of their connected individual discrete system elements. DESim's modeling methodology focuses on the modeling of each individual element and utilizes an advance and await mechanism for simulation execution. Thus DESim's modeling methodology does not rely on predetermined events and event execution times for the start of modeled system element tasks. This methodology readily lends itself to the efficacy of computer architectural modeling and provides a platform to accurately model physical system elements, element interactions, and system-wide occupancy and contention. In this paper, we discuss DESim's implementation approach and provide a detailed example of a modeled switching fabric to highlight the power, scalability, and ease of use of DESim's modeling methodology in computer architectural simulations.**

I. Introduction

In this paper we present DESim; a framework and methodology for detailed computer architectural simulation. DESim is a simulation tool that is implemented as a lightweight library that provides a robust discrete element simulation framework. DESim provides researchers the tools and ability to rapidly produce detailed computer architectural simulation models in an environment with low levels of abstraction from physical systems. This is accomplished in DESim by explicitly modeling the system as a collection of its individual discrete functional elements, which is a unique approach as compared to other mainstream event driven simulation engines like those found in modern mainstream computer architectural simulators, like GEM5 [1] and Multi2Sim [2].

In comparison, DESim's modeling methodology approach focuses on the modeling of each individual element and utilizes an advance and await mechanism for simulation execution. Thus, DESim's modeling methodology does not rely on predetermined events and event execution times for simulation execution as found in common event driven simulation engines. The nature of DESim's modeling of individual elements readily lends itself to computer architectural modeling and provides a platform to accurately models physical system elements, element interactions, and system-wide occupancy and contention.

DESim's modeled discrete functional elements closely resemble the real physical hardware which allows researchers to easily build the modeled system as a collection of its smaller elements and does not require researchers to think in highly abstract terms. Additionally, DESim's modeling mechanism scales well with simulation model size, promotes reuse of modeled elements in future computer architectural simulators, and allows for easier translation of the higher-level simulator based source code to lower-level hardware description languages, such as VHDL [3] and SystemC [4].

The main contributions of this paper are:

- We present a detailed discussion, with pseudocode, of the implementation methodology of the DESim framework and provide a ready-made and fully working implementation of the DESim library for researchers to download and utilize. Our implementation of the DESim library is made available as free software and can be found on GitHub.
- We extend DESim's implementation methodology and correct functional issues that occur due to the recent additions of stack protection, pointer mangling, and source fortification in the Linux software stack. Our implementation is compatible with modern 32bit and 64bit x86 Linux distributions.
- We provide a detailed example describing the methodology of usage of the DESim framework for computer architectural functional and power simulations. Our detailed example provides the reader with the knowledge needed to effectively utilize the DESim library in their own research and highlights the power, scalability, and ease of use of DESim.
- We discuss DESim's parallel processing potential and provide new directions for future research regarding parallelizing the DESim library.

The rest of this paper is organized as follows. Section II provides details and pseudocode regarding the implementation of the DESim framework and its pertinent software architectural and mechanical elements. Section III provides

a detailed switching fabric example and pseudocode describing the methodology of usage of the DESim framework for computer architectural modeling. Pertinent aspects, such as functional and power simulation modeling are discussed. Section IV summarizes related work to our own. Section V covers future work where we discuss new directions in regards to parallelizing the DESim framework and, finally, section VI concludes the paper.

II. DESim Framework Implementation Methodology

The DESim framework is implemented as a standalone and lightweight library that can be readily included and utilized in computer architectural simulation research efforts. Currently, DESim is coded in the ANSI C language and supports operation in 32bit and 64bit Linux x86 environments.

Externally, DESim comprises eight user level functions that allow the user to rapidly create and model simulation objects and control the flow of execution in the modeled system during simulation execution. The user level functions are written such that the user is not burdened with simulation engine details. DESim's simple interface provides a high level of efficacy in programmability and allows the user to focus solely on the important aspects of the individual elements being modeled. Details regarding the inner workings of the user level functions are presented in this section and descriptions of their use are presented in Sec. III.

Internally, DESim comprises the objects and mechanisms necessary to model complete computational systems as a collection of its connected individual discrete system elements. Each individual discrete element modeled is represented by a single user provided function, which defines the discrete element's tasking. During simulation execution, simulated elements await (sleep) until they are advanced (woken up) and given work to perform. When a discrete element is advanced the element will wake up and try to process its assigned work until success. Once the element completes its task, it will then advance the next element and give it work to perform. While an element is working the element may assess a latency commensurate with the work it must perform. To access latency the element pauses until that latency (number of cycles) has passed and then resumes from where it previously paused.

The element's assessed latency provides the mechanism to automatically model the occupancy of that element as no other work can be performed by the element during that time. Contention is automatically modeled as simulation elements must compete for a given resource. Individual elements must stall as they wait for access to a particular resource. These stalls result in longer access latency as elements wait for modeled hardware resources to become available. In the following subsections we provide discussion and present the details of the implementation of DESim's framework.

Algorithm 1 Globals

```
1:  globals
2:      if wordsize == 64 then
3:          typedef long int _jmp_buf[8];
4:      else if x86_64 then
5:          typedef long long int _jmp_buf[8];
6:      else
7:          typedef int _jmp_buf[6];
8:      end if
9:
10:     struct list{
11:     int num_elements;
12:     int size;
13:     int head;
14:     int tail;
15:     void ** elem; };
16:
17:     struct eventcount{
18:     list*ctxlist;
19:     unsigned long long count; };
20:
21:     struct context{
22:     jmp_buf buf;              ▷ Buffer for CPU registers
23:     unsigned long long count;
24:     void (*start)(void);      ▷ User defined function
25:     char* stack;              ▷ This context's stack
26:     int stacksize; };
27:
28:     eventcount* etime = NULL; ▷ Global eventcount
29:     list* gctxlist = NULL;    ▷ Global context list
30: end globals
```

A. Contexts and Eventcounts

The DESim simulation engine can be broken down into two major functional software components, which are contexts and eventcounts [5]. Pseudocode describing our implementation of contexts and eventcounts is shown in Alg. 1. We implement contexts and eventcounts as structures and create and initialize them as required to model the desired system.

In DESim, contexts are individually executable objects that represent the discrete elements modeled by the user. The context structure comprises a jump buffer, count, function pointer, stack pointer, and stack size. The jump buffer is a primitive data type that is utilized by our implemented `setjmp()` and `longjmp()` functions, see Sec. II-D. The jump buffer's data type and size is architecture dependent. In the 32bit and 64bit Linux x86 environments the jump buffer's data type is either of type `int`, `long int`, or `long long int`. Determination of the correct data type based on the Linux environment is shown at the top of Alg. 1. The context's count is used to synchronize the context with an eventcount's state. The context's function pointer is assigned the address of the user's provided function describing this individual element's tasking. The stack pointer points to an allocated region of

Algorithm 2 Initialization

```
1: procedure EVENTCOUNT_INIT(void)
2:     eventcount* ec ← NULL;
3:     ec ← (eventcount*)malloc(sizeof(eventcount));
4:     ec->count ← 0;
5:     ec->ctxlist ← list_create();
6:     return ec;
7: end procedure
8:
9: procedure CTX_INIT((*func)(), unsigned size)
10:    context* ctx ← NULL;
11:    ctx ← (context*)malloc(sizeof(context));
12:    ctx->count ← etime-> count;
13:    ctx->stack ← (char*)malloc(size);
14:    ctx->stacksize ← size;        ▷ Stack overflow check
15:    ctx->start ← func;            ▷ User defined function
16:    ctx->buf[ip] ← context_start();
17:    ctx->buf[sp] ← stack_top_ptr;
18:    insert_list_item(gctxlist, 0, ctx);
19:    return
20: end procedure
21:
```

memory of user provided stack size for each context. Each context's stack is unique, resides in user memory space, and contains that context's execution data.

Eventcounts are objects that provide an abstract mechanism to enforce a relative ordering of events that have yet to occur. In DESim, eventcounts comprise a count, which is used as an incrementer, and a list structure for storage of contexts. The eventcount's count records the number of times the eventcount has been advanced (incremented). Contexts await the advancement of eventcounts and once the eventcount's and context's count are equal the context can be scheduled to run. During execution newly awaiting contexts are added, in count order, to the eventcount's context list. Typically, each context will require at least one unique eventcount assigned to it. There is one global eventcount, titled etime, that is associated with the global cycle count.

B. Context and Eventcount Initialization

Pseudocode showing the initialization approach for eventcounts and contexts is shown in Alg. 2.

Prior to simulation execution the global context list and etime, shown in Alg. 1, must be initialized by the user. DESim provides a single function, `desim_init()`, that the user calls to accomplish this. Subsequently, all user defined eventcounts and contexts are created utilizing the provided `eventcount_init()` and `ctx_init()` functions.

Eventcount initialization is straightforward and comprises the allocation of the eventcount with the use of `malloc()`, the configuration of the eventcount's count, and the configuration of the eventcount's context list. At initialization time the eventcount's count is set to zero and an empty context list is allocated and assigned to the eventcount's context list pointer.

Context initialization comprises the allocation of the context itself with the use of `malloc()`, configuration of the context's count, allocation of the context's stack with use of `malloc()`, assignment of the stack size, assignment of the user's provided function to the context's start function pointer, manipulation of the context's jump buffer instruction and stack pointers, and finally insertion of the context itself into the global context list. At initialization time the context's count is assigned the global cycle count which is etime's count. This allows contexts to be created and destroyed at any time, both before and during simulation execution. The start function is provided by the user and is assigned to the context's start function pointer. The start function embodies the functionality of the individual discrete element this context will simulate. Examples further demonstrating the relationship between the context and its individual element are further provided in Sec. III.

The context's jump buffer is raw at initialization. Thus, we must assign a starting instruction pointer and stack pointer by hand to give the context our desired starting position. At initialization other CPU registers can be ignored because they will be obtained the first time `setjump()` is called. The pseudocode shows an instruction pointer assignment as the head of the `context_start()` function. The makeup of the `context_start()` function is shown in Alg. 6. On initial execution of each context the `context_start()` function is the first place each context jumps to. Immediately after jumping to `context_start()` the context's `start()` function is called which places program execution at the head of the user's provided function. The pseudocode also shows a stack pointer assignment as the top of the allocated stack. For the 32bit and 64bit Linux x86 environments the top of the stack is calculated as shown in Equ. 1.

$$stack\_top\_ptr = stack\_ptr + stack\_size - sizeof(int) \quad (1)$$

The assignment of the instruction and stack pointers to the context's jump buffer is architecture dependent and must be accounted for at time of compilation. In the 32bit Linux x86 environment the instruction pointer and stack pointers are assigned to jump buffer positions *five* and *four* respectively. In the 64bit Linux x86 environment the instruction and stack pointers are assigned to jump buffer positions *seven* and *six* respectively.

C. Context Scheduling and Context Switching

Pseudocode showing the mechanisms required for context scheduling and context switching is shown in Alg. 3 and Alg. 4.

During execution each context resides in either an await, ready to run, or running state. In the single threaded version of DESim only one context is ever in the running state at a time. Transitions between these states are accomplished with use of the `advance()`, `await()`, and `pause()` functions. A running context executes its assigned tasks and advances one or more eventcounts as a product of its work. By advancing an eventcount, the designated eventcount's count is incremented and the eventcount's local context list is traversed. All contexts

Algorithm 3 Context Scheduling

```
1: procedure ADVANCE(eventcount* ec)
2:     context* ctx ← NULL;
3:     ec->count ← ec->count + 1;
4:     for i ← 0 to ec->ctxlist->num_elements − 1; do
5:         ctx ← get_ptr_to_list_item(ec->ctxlist, i);
6:         if ctx and ctx->count == ec->count then
7:             ctx->count ← etime->count;
8:             ctx ← dequeue(ec->ctxlist);
9:             enqueue(gctxlist, ctx);          ▷ Global ctxlist
10:        else
11:            break;
12:        end if
13:    end for
14:    return
15: end procedure
16:
17: procedure AWAIT(eventcount* ec, count value)
18:    context* ctx ← NULL;
19:    if ec->count >= value then
20:        return;
21:    end if
22:    for i ← 0 to ec->ctxlist->num_elements do
23:        ctx ← get_ptr_to_list_item(ec->ctxlist, i);
24:        if !ctx or (ctx and value < ctx->count) then
25:            ctx ← dequeue(gctxlist);         ▷ Global ctxlist
26:            ctx->count ← value;
27:            insert_list_item(ec->ctxlist, i, ctx);
28:            break;
29:        end if
30:    end for
31:    context_switch(context_select());
32: end procedure
33:
34: procedure PAUSE(count value)
35:    await(etime, etime->count + value);
36:    return
37: end procedure
```

Algorithm 4 Context Switching

```
1: global variables
2:     context* current_ctx ← NULL;
3:     context* last_ctx ← NULL;
4: end global variables
5:
6: procedure CONTEXT_SELECT(void)
7:     current_ctx ← get_ptr_to_list_item(gctxlist, 0);
8:     if !current_ctx then
9:         current_ctx ← dequeue(etime->ctxlist);
10:        if !current_ctx then
11:            sim_end();          ▷ All ctx awaiting eventcount
12:        end if
13:        insert_list_item(gctxlist, 0, current_ctx);
14:    end if
15:    etime->count ← current_ctx->count;
16:    return current_ctx;
17: end procedure
18:
19: procedure CONTEXT_SWITCH(context* current_ctx)
20:    if !last_ctx or !setjmp(last_ctx->buf) then
21:        last_ctx ← current_ctx;
22:        longjmp(current_ctx->buf, 1);
23:    end if
24:    return;
25: end procedure
```

with a count equal to the eventcount's count are removed from the eventcount's local context list, assigned the current cycle count (provided by etime), and placed into the global context list as contexts now in the ready state.

After completion of its tasks, the running context then transitions to the await state by use of the `await()` function. Transitioning to the await state is accomplished by yielding to a ready context. During the process of yielding, the context will assign itself a count in which it will await, remove itself from the global context list, insert itself into the designated eventcount's local context list (in count order) as an awaiting context, and hand over control to the next ready context. During execution of its task the running context may also assess a latency with the use of the `pause()` function. This also transitions the running context into the await state, however assessing a latency associates the context with etime where it will now await a future cycle count instead of an advancement of a particular user created eventcount.

The next ready context is selected with the `context_select()` function. As mentioned before, ready contexts reside in the global context list. Selecting the next ready context simply requires obtaining a pointer to the context stored at the head of the global context list. Prior to the completion of selecting the next ready context, etime's count is updated to reflect the current cycle count. Ready contexts stored in the global context list all share the same cycle count and are to run this cycle. When there are no longer any ready contexts in the global context list all activity to be performed in a given cycle has completed and simulation can now move forward into a future cycle. In DESim, the simulated cycle count does not linearly increase and execution will time warp to the cycle of the next ready context in etime's context list. Advancing to a future cycle is then performed by dequeuing a context from the head of etime's local context list, inserting that context into the empty global context list, assigning the awaiting context's count to etime's count, and returning a pointer to that context. Note that if both the global context list and etime's local context list are empty simulation has ended.

After the next context is selected the running context yields to the next context and switches execution with use of the `context_switch()` function. Performing the switch results

in a jump via the use of DESim's hand coded `setjmp()` and `longjmp()` assembly functions, see Sec. II-D. During the process of yielding, the context's jump buffer is updated to the current position and then a jump is made to the previously stored position of the next context. This process repeats as contexts alternate between the running and await states. On the initial jump the context will jump to the head of the `context_start()` function, see Alg. 6. This position was manually entered into the context's jump buffer during initialization and provides a return point should the context need to return.

Global context pointers for the last context and current context are utilized in the context switching mechanism. The last context pointer is used to ensure that the jump buffer of the yielding context is updated by `setjmp()`. Use of the current context pointer is not necessary, however the current context pointer provides a way to query data regarding the running context at any time.

D. Setjmp and Longjmp

The standard Libc package provides primitives for the jump buffer data type and also provides versions of the `setjmp()` and `longjmp()` functions. The data type of the jump buffer and composition of the `setjmp()` and `longjmp()` functions are architecture dependent and each has a different implementation for the 32bit and 64bit x86 Linux environments. The `setjmp()` function works to store away the registers of the CPU, stack pointer, and instruction pointer in the provided jump buffer. The `longjmp()` function works to restore those saved values and then proceeds to jump to the location of the instruction pointer saved by the call to `setjmp()`. When returning to the position of the last `setjmp()` from a `longjmp()`, `setjmp()` will return the value passed as the second argument to `longjmp()`. This allows for a test to determine if a `setjmp()` is being called for the first time or if program execution is returning to `setjmp()` from a `longjmp()`.

In DESim we have coded `setjmp()` and `longjmp()` by hand as standalone assembly files. Pseudocode providing detail regarding our versions of the `setjmp()` and `longjmp()` functions is shown in Alg. 5. In DESim `setjmp()` and `longjmp()` are called like any regular function with the caveat that the calling function must account for the target architecture during compilation. For brevity, we only show the 64bit x86 implementations. For 32bit x86 implementations please refer to the DESim source code.

Our implementation of `setjmp()` and `longjmp()` corrects functional issues that occur due to the recent additions of stack protection, pointer mangling, and source fortification in the Linux software stack. Usage of the standard Libc `setjmp()` and `longjmp()` will render this simulation methodology non-functional.

E. The Simulation State

Pseudocode showing how to place DESim in the simulation state is shown in Alg. 6. After properly creating and initializing all DESim related globals, user eventcounts, and user contexts, the computer architectural model can be placed into the simulate state by calling the `simulate()` function. In the simulate state, DESim first turns the user's main function into a unique context with use of the `context_sim()` function. After which, DESim will switch away to the first ready context in the global context list. On the initial execution of each context, the physical switching of the contexts results in a jump to the head of the `context_start()` function. At the end of simulation DESim will switch back to the user's main function with use of the `sim_end()` function. Prior to returning execution to the user, DESim performs cleanup activities where it destroys eventcounts, contexts, and other DESim related objects.

Algorithm 5 Jumping

```
1:  procedure SETJMP(jmp_buf buf)
2:      .section .text
3:      .globl setjmp
4:      .type setjmp, @function
5:      setjmp :
6:      mov   %rbx, (%rdi)            ▷ Store callee registers
7:      mov   %rbp, %rax
8:      mov   %rax, 0x8(%rdi)
9:      mov   %r12, 0x10(%rdi)
10:     mov   %r13, 0x18(%rdi)
11:     mov   %r14, 0x20(%rdi)
12:     mov   %r15, 0x28(%rdi)
13:     lea   0x8(%rsp), %rdx         ▷ Get stack pointer
14:     mov   %rdx, 0x30(%rdi)        ▷ Sore stack pointer
15:     mov   (%rsp), %rax            ▷ Get inst pointer
16:     mov   %rax, 0x38(%rdi)        ▷ Store inst pointer
17:     mov   %rax, %rax
18:     ret
19: end procedure
20:
21: procedure LONGJMP(jmp_buf buf, int val)
22:     .section .text
23:     .globl longjmp
24:     .type longjmp, @function
25:     longjmp :
26:     mov   0x8(%rdi), %r9          ▷ Restore callee registers
27:     mov   0x10(%rdi), %r12
28:     mov   0x18(%rdi), %r13
29:     mov   0x20(%rdi), %r14
30:     mov   0x28(%rdi), %r15
31:     mov   0x30(%rdi), %r8         ▷ Restore stack pointer
32:     mov   0x38(%rdi), %rdx        ▷ Restore inst pointer
33:     mov   (%rdi), %rbx
34:     mov   %esi, %eax              ▷ Set return val
35:     mov   %r8, %rsp
36:     mov   %r9, %rbp
37:     jmpq  *%rdx
38: end procedure
```

Algorithm 6 Simulate

```
1: global variables
2:     jmp_buf main_ctx;
3: end global variables
4:
5: procedure MAIN(void)                    ▷ User's main function
6:     desim_init();                  ▷ Init DESim global elements
7:     model_init();          ▷ Init all user simulation elements
8:     simulate();                        ▷ Enter simulation state
9:     finalize();              ▷ Simulation has ended, clean up
10:    return;
11: end procedure
12:
13: procedure SIMULATE(void)
14:    if !context_sim() then
15:        context_switch(context_select());
16:    end if
17:    clean_up();            ▷ Free library objects and memory.
18:    return                      ▷ Return to user defined main.
19: end procedure
20:
21: procedure CONTEXT_SIM(void)
22:    return setjmp(main_context);
23: end procedure
24:
25: procedure CONTEXT_START(void)
26:    (*current_ctx->start)();  ▷ Call user defined func
27:    sim_end();                      ▷ User defined func returned
28:    return;
29: end procedure
30:
31: procedure SIM_END(void)
32:    longjmp(main_context, 1);
33:    return
34: end procedure
```

III. DESim Simulation Modeling Methodology

Using DESim to create computer architectural functional and power models is straightforward. The user establishes a generic control function encapsulating the tasks that a particular type of simulation element will perform. Thus, the computer architectural model, in whole, is represented as the collection of these simulation elements cooperatively working together.

A simple usage example of this methodology has been previously established in [6]. In the example, two simulation objects are represented as a producer/consumer pair. During execution the producer performs work, provides data to the consumer, advances the consumer, and then awaits. After advancement the consumer wakes up, performs work, provides data to the producer, and then awaits. In this case, the producer awaits its eventcount and advances the consumer's event count. Likewise, the consumer awaits its eventcount and advances the producer's event count. The cycle repeats until the simulation ends. This is a typical representation of execution flow for all individual simulation elements. In a large computer architectural model, long chains of producer/consumer pairs represent the interdependence between discrete hardware elements.

However, the simple producer/consumer example alone does not provide the level of detail required to fully grasp the effective usage of DESim. Therefore, we build upon this previous example and present a switching fabric model as a detailed usage example of DESim's modeling methodology. In our example switching fabric model we demonstrate how to use DESim for modeling of system elements and highlight the ease of use and power of DESim's modeling methodology.

A. Functional Simulation Methodology

Pseudocode showing our detailed switching fabric model is shown in Alg. 7. In our example, we model a switch and crossbar with a user defined number of ports and virtual lanes. Our switch pseudocode is written with the intent of clearly relaying DESim's modeling methodology. More concise and sophisticated implementations are possible. The model consists of a main control function `switch_ctrl()` where all derivative switch related tasks are encapsulated.

Variables pertinent to the individual switch model are located along the top of the main control function. These variables and other newly created variables are all stored in the individual context's stack. Any number of switches can be created utilizing this single main control function. Therefore, we assign a `PID` to each switch for dereferencing the correct switch structure and eventcount during execution. The `await()` function takes as an argument a count to assign to the context as it waits. So, a step variable is needed to manage this interaction and is initialized with the value of one. This causes the switch to enter the await state at the beginning of the switch's main execution loop where it then awaits advancement by one or more connected system elements.

Arguably, one of the most powerful features of the DESim framework is the ability to easily model interactions in the sub-clock domain. In relation to computer architectural simulations, this provides an easy way to model arbitration in the system. We provide an example of this in our switch model. After advancement by one or more connected elements the switch then enters the sub-clock domain. During which all elements seeking to advance the switch have completed their advancement and entered the await state. The switch can then perform arbitration and scheduling functions with complete knowledge of all its advancing elements. In modeling the sub-clock domain, the cycle count can be divided any number of times with the use of macros intended to adjust the cycle time. In this example our macros, `ENTER_SUB_CLOCK` and `EXIT_SUB_CLOCK` adjust the cycle count such that two cycles represent one cycle for the system at large. The sub-clock domain is then modeled on the odd cycle count which is the half cycle to the system.

After returning to the regular clock domain, the switch then charges latency for its work with the `P_PAUSE` macro. The switch will stop and then resume from this point later

Algorithm 7 Switch Control

```
1: globals
2:     #define CYCLE etime->count>>1
3:     #define P_PAUSE(p_delay) \
4:         pause((p_delay)<<1)
5:     #define ENTER_SUB_CLOCK \
6:         if(!(etime->count & 0x1)) pause(1);
7:     #define EXIT_SUB_CLOCK \
8:         if (etime->count & 0x1) pause(1);
9:     int switch_pid = 0;
10:    eventcount ** sw_ec;
11:    struct switch_t ** sw;
12: end globals
13:
14: procedure SWITCH_CTRL(void)
15:    int my_pid = switch_pid++;
16:    count step = 1;
17:    packet* net_packet = NULL;
18:    loop
19:        await(&sw_ec[my_pid], step);
20:        calc_passive_power(&sw[my_pid], CYCLE);
21:        ENTER_SUB_CLOCK
22:        xbar_link(&sw[my_pid]);
23:        EXIT_SUB_CLOCK
24:        P_PAUSE(&sw[my_pid]->latency);
25:        for i ← 0 to sw[my_pid]->num_ports − 1 do
26:            if xbar_link_success(&sw[my_pid], i) then
27:                sw[my_pid]->num_links++;
28:                net_packet ← dequeue(&sw[my_pid], i);
29:                enqueue(&sw[my_pid], i, net_packet);
30:                advance(&sw_io_ec[
31: xbar_out(&sw[my_pid], i)];
32:            end if
33:        end for
34:        step += sw[my_pid]->num_links;
35:        calc_active_power(&sw[my_pid],
36: &sw[my_pid]->num_links, CYCLE);
37:        sw[my_pid]->num_links = 0;
38:        switch_update_state(&sw[my_pid]);
39:    end loop
40:    return;                          ▷ Should never return
41: end procedure
```

after the specified number of cycles pass. The charged latency automatically provides a mechanism to model the occupancy of the switch. Upon resuming, the switch then records the number of successfully formed links, moves data from the specified input ports to the specified output ports, and advances the specified output port I/O controller. If resources concerning one or more of the switch's I/O controllers are saturated, the switch must appropriately stall until those resources become available. The switch's stalls automatically model contention in regards to those resources.

Prior to returning to the top of the switch's main execution loop the switch resets its state and prepares for the arrival of new work. The switch's step value must be incremented via the number of links made. This accounts for the number of advancements and the work performed that cycle. In the case that the switch fails to service any requests, the main execution loop must continue to run until all outstanding request are serviced and the step variable's value returns to one larger than the switch's eventcount value.

B. Power Simulation Methodology

In our example switch model we show the methodology behind creating accurate power simulation models. This is accomplished by calculating the individual element's passive and active power expenditures throughout execution. Each element has a finite number of possible derivative states it can enter. In our modeled switch example, individual switch elements are either in the run/ready state or are in the await state. Calculations determining the expenditure of power during these two states is performed with the `calc_passive_power()` and `calc_active_power()` functions.

The await state is straightforward and represents the period of time the element passively utilizes/leaks power. After being advanced the switch wakes up and calculates the passive/leaked power usage over the period of time the switch was awaiting. This is accomplished by utilizing an assigned value representing the passive power used per cycle. Upon exiting simulation one final calculation must be made to reconcile the difference between the element's last await and the end of simulation.

During the run/ready state several derivative outcomes may occur that would influence the amount of power utilized by the element. This necessitates a user provided state based power profile for the modeled element. The power profile maps each possible outcome to an assigned value representing the active power used in that state. The nature of the switch's main execution loop makes it easy to determine the possible states of the switch model during the run/ready state. In our example, we use the number of links formed while in the run/ready state to estimate the level of effort performed by the switch.

When added together the passive and active power usage determines the overall power expenditure for the modeled element. The expenditure for each modeled element can be aggregated to determine the power expenditure for the entire system or portions of the system thereof. Impacts to power from proposed architectural changes can then be accurately modeled and taken into account by researchers.

IV. Related Work

There is a long and diverse history of related work concerning discrete event simulation covering a broad spectrum of methodologies and techniques. Information regarding many of these methods and techniques can be found over the course of relevant surveys; examples of some relevant surveys include [7], [8], and [9]. In general, related work to our own falls in the category of discrete event simulation techniques intended for use in the development of computer architectural simulations.

Thus for brevity and relevance, we limit our related work section to a summary of the historical usage of the modeling methodology presented in this paper and other discrete event simulation techniques used in current mainstream computer architectural simulation systems.

Historically, DESim's modeling methodology has been used in at least one [10] publicly known computer architectural simulation system, but has previously been little discussed. Stanford's FlashLite simulator is a detailed system-level simulator that can run benchmarks from the SPLASH benchmark suite [11], enable the verification of memory system protocols, be used in analysis of system performance, and help in identification of architectural bottlenecks. The FlashLite simulator was used in the research supporting the system design and verification of the Stanford FLASH system—a physical system built by Stanford. At the heart of FlashLite is an earlier predecessor to the modeling methodology presented in this paper that was introduced by Earl Killian of MIPS Technologies, Inc and called "The Threads Package". In FlashLite, simulated system element models were structured identically to the hardware found in FLASH. In FlashLite's modeling methodology, individual contexts are referred to as "FlashLite Threads" and, as mentioned before, a description of the await and advance modeling methodology is presented in [6]. In comparison, the methodology implemented in FlashLite is functionally equivalent to the methodology implemented within DESim. Meaning that DESim's implementation methodology preserves the interfaces and functional mechanisms of its prior implementations. However, DESim extends its predecessor's implementation and provides a modernized standalone library that includes corrections to functional issues that occur due to Linux software stack protection, pointer mangling, and source fortification, see Sec. II-D. Without these extensions the issues encountered with usage in modern Linux distributions render previous implementations of the methodology non-functional.

Examples of directly related work regarding other discrete event simulation techniques used in current mainstream computer architectural simulation systems can be found in GEM5 [1], Multi2Sim [2], Ruby [12] and their derivative computer architectural simulation systems, such as Gem5-GPU [13], and FusionSim [14]. GEM5, Multi2Sim, and their derivative computer architectural simulation systems each employ a discrete event simulation tool that utilizes a similar technique. In each, the discrete event simulation system works by scheduling and executing a predetermined callback function at a specified time (cycle count). In essence, the discrete event simulation system's scheduler will call the function passed to it when the number of cycles provided by the developer transpires. Ruby employees a slightly different technique from GEM5 and Multi2sim. In Ruby messages are enqueued in buffers linking modeled system elements together. The buffers themselves, comprise variable latency and bandwidth properties. Simulation execution proceeds by invoking a callback function for the next scheduled event on a given event queue. In each of these approaches researchers endeavor to model individual system elements as a collection of individual functions and appropriate execution timings.

In comparison to the modeling methodologies incorporated in GEM5, Multi2Sim, Ruby and their derivative computer architectural simulation systems, DESim's modeling methodology focuses on the modeling of each individual element as a single context that encapsulates its required functional properties. Simulation execution is carried out with an advance and await mechanism. Thus, DESim's modeling methodology does not utilize predetermined functions and execution times for the start of modeled system element tasks. This is a unique approach as compared to the approach employed in in GEM5, Multi2Sim, Ruby and their derivative computer architectural simulation systems. With DESim, modeled functional elements closely resemble the real physical hardware and does not require researchers to think in more abstract terms allowing them to focus on the important aspects of the individual elements being modeled at hand.

V. Future Work

Future work includes our efforts to parallelize the DESim framework. In DESim, all system elements are modeled as individually executable contexts and in a large computer architectural model it is common to have hundreds or thousands of contexts. DESim's approach of modeling these individually executable contexts readily lends itself to parallelization. In general, all contexts can perform their assigned tasks, separately, as individual threads in a parallelized environment. However, during simulation execution a call to `advance()`, `await()`, and global cycle count increases would briefly introduce serialization points between one or more of the running contexts.

Our future plans are to perform additional research related to implementing an effective thread management schema within DESim. Contexts may run as individual threads and can individually perform the work of scheduling and launching new contexts as the running context halts at the end of an await. Additionally, we plan to perform research regarding mitigating the impacts to speedup occurring at DESim's serialization points. We believe that a robust thread pool model and the inclusion of a lock-free list data structure and atomic count manipulations will theoretically result in measurable speedups over the single-threaded approach. Based on the results of our related work, it appears that it is not publicly known if a parallel implementation of the DESim library currently exists or not.

VI. Conclusion

In this paper we present DESim; a framework and methodology for detailed computer architectural simulation. We provide in-depth details regarding the DESim framework software architectural makeup and its implementation methodology. Additionally, we provide details and a demonstration of DESim's usage in the implementation of functional and power simulation models with our detailed switching fabric example. Our example switching fabric demonstrates how to use DESim for modeling of computer architectural system elements and

highlights the ease of use and power of DESim's modeling methodology. Finally, we discuss future work and propose new directions in research efforts regarding the parallelization of the DESim framework.

A standalone version of the DESim framework and the examples presented in this paper are made available for public use as free software for future research purposes. The most current version of DESim can be found on GitHub.

REFERENCES

[1] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[2] NEU, "Multi2sim a heterogeneous system simulator," accessed: Jan-23-2017. [Online]. Available: http://www.multi2sim.org/

[3] IEEE, "Ieee standard for verilog hardware description language," accessed: Oct-6-2017. [Online]. Available: http://ieeexplore.ieee.org/document/1620780/

[4] Accellera, "Systemc," accessed: Oct-6-2017. [Online]. Available: http://www.accellera.org/downloads/standards/systemc

[5] D. P. Reed and R. K. Kanodia, "Synchronization with eventcounts and sequencers," *Commun. ACM*, vol. 22, no. 2, pp. 115–123, Feb. 1979. [Online]. Available: http://doi.acm.org/10.1145/359060.359076

[6] M. Heinrich, D. Ofelt, M. A. Horowitz, and J. Hennessy, "Hardware/software co-design of the stanford flash multiprocessor," *Proceedings of the IEEE*, vol. 85, no. 3, pp. 455–466, Mar 1997.

[7] S. Robinson, "Discrete-event simulation: from the pioneers to the present, what next?" *Journal of the Operational Research Society*, vol. 56, no. 6, pp. 619–629, Jun 2005.

[8] V. yee Vee and W. jing Hsu, "Parallel discrete event simulation: A survey," Tech. Rep., 1999.

[9] F. J. Kaudel, "A literature survey on distributed discrete event simulation," *SIGSIM Simul. Dig.*, vol. 18, no. 2, pp. 11–21, Jun. 1987.

[10] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy, "The stanford flash multiprocessor," in *25 Years of the International Symposia on Computer Architecture (Selected Papers)*, ser. ISCA '98. New York, NY, USA: ACM, 1998, pp. 485–496.

[11] S. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *Proc. of the 22nd International Symposium on Computer Architecture*, June 1995.

[12] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (gems) toolset," *SIGARCH Comput. Archit. News*, vol. 33, no. 4, pp. 92–99, Nov. 2005.

[13] J. Power, J. Hestness, M. S. Orr, M. D. Hill, and D. A. Wood, "gem5-gpu: A heterogeneous cpu-gpu simulator," *IEEE Computer Architecture Letters*, vol. 14, no. 1, pp. 34–36, Jan 2015.

[14] V. Zakharenko, T. Aamodt, and A. Moshovos, "Characterizing the performance benefits of fused cpu/gpu systems using fusionsim," in *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2013, pp. 685–688.