2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th Intl. Conference on Data Science and Systems

A Fast Discrete Event Driven Simulation Methodology For 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*— In this paper we introduce a fast discrete event driven simulation methodology, called KnightSim, that is intended for use in the development of future computer architectural simulations. KnightSim extends an older event driven simulation library by (1) incorporating corrections to functional issues that were introduced by the recent additions of stack protection, pointer mangling, and source fortification in the Linux software stack, (2) incorporating optimizations to the event engine, and (3) introducing a novel parallel implementation. KnightSim implements events as independently executable x86 "KnightSim Contexts". KnightSim Contexts comprise a mechanism for fast context execution and automatically model occupancy and contention, which readily lends itself to use in computer architectural simulations. We present the implementation methodologies of KnightSim and Parallel KnightSim with a detailed performance analysis. Our performance analysis makes direct comparisons between KnightSim, Parallel KnightSim, and the discrete event driven simulation engines found in three different mainstream computer architectural simulators. Our results show that on average KnightSim achieves speedups of 2.8 to 11.9 over the other discrete event driven simulation engines. Our results also show that on average Parallel KnightSim can achieve speedups over KnightSim of 1.78, 3.30, 5.84, and 9.16 for 2, 4, 8, and 16 threaded executions respectively.**

I. Introduction

In this paper we introduce a fast discrete event driven simulation methodology, called KnightSim, that is intended for use in the development of future computer architectural simulations. KnightSim extends an older proven event driven simulation methodology known as "The Threads Package". The Threads Package has previously been used in at least two publicly known computer architectural simulators [1], [2]. Our extensions to The Threads Package are summarized as:

- A novel implementation with fixes to several functional issues that have occurred due to the recent additions of stack protection, pointer mangling, and source fortification in the Linux software stack. Our implementation is coded in the ANSI C language and supports operation in 32bit and 64bit Linux x86 environments.
- An optimized event driven simulation engine. Our implementation of KnightSim achieves a measurable speedup over the preceding version of the methodology.
- A novel parallel implementation that automatically parallelizes event execution at the cycle level. Our parallel implementation, called Parallel KnightSim, is capable of achieving further speedups over the single threaded version.

KnightSim comprises a unique discrete event driven simulation methodology as compared to those found in modern mainstream computer architectural simulators, like GEM5 [3] and Multi2Sim [4]. KnightSim implements events as independently executable x86 "KnightSim Contexts". By design, KnightSim Contexts encapsulate all of the functionality and interfaces associated with a single target simulated system element in an individually executable package. This implementation methodology enjoys several benefits from this approach. First, occupancy and contention, which have been proven to be a critical determinant of system performance [5], are automatically modeled by KnightSim Contexts. Other simulation methodologies, like those of Gem5 and Multi2Sim, do not do this and require additional events, state flags, and levels of abstraction to achieve a realistic occupancy and contention model. Second, executing a KnightSim Context only requires a long jump, see Sec. III-A. This mechanism is faster as compared to scheduling and running an event's call-back function because a KnightSim Context's stack is not created and torn down each time the context is executed. Finally, KnightSim Context execution can be performed in parallel because each KnightSim Context is independently executable in a multithreaded environment. These properties make KnightSim a promising tool for use in the development of computer architectural simulations.

The main contributions of this paper are:

- We present a detailed discussion, with pseudocode, of our implementation of KnightSim and Parallel KnightSim.
- We present the results of a performance analysis of KnightSim, Parallel KnightSim and three different event driven simulation methodologies that are widely in use today.
- We provide a ready-made and fully working implementation of KnightSim and Parallel KnightSim, with

usage examples, for researchers to download and utilize. KnightSim and Parallel KnightSim are made available as free software and can be found on GitHub.

II. 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 a few comprehensive and relevant surveys [6], [7], [8]. In general, related work to our own falls into the category of discrete event simulation methodologies intended for use in the development of computer architectural simulations. Thus for brevity and relevance, we limit our related work to the methodologies used in current mainstream computer architectural simulation systems.

The implementation methodology from which KnightSim inherits its base functionality from is called The Threads Package and has been used in at least the FlashLite and M2S-CGM computer architectural simulation systems [1], [2]. However despite its prior usage, implementation details regarding The Threads Package itself have previously been little discussed. Our implementation of KnightSim preserves the interfaces of The Threads Package making them functionally equivalent to each other. However, KnightSim incorporates a completely redesigned and optimized implementation that results in (1) fixes to functional issues that otherwise render the methodology non-functional in modern Linux distributions, (2) a significant performance enhancement, and (3) a novel parallelized implementation that is further still capable of higher levels of performance.

Examples of directly related work regarding other discrete event simulation methodologies used in current mainstream computer architectural simulation systems can be found in GEM5 [3], Multi2Sim [4], Ruby [9], and their derivative computer architectural simulation systems, such as Gem5-GPU [10] and FusionSim [11]. Each of these computer architectural simulation systems employ a discrete event simulation tool that utilizes a similar technique. In each, the discrete event simulation engine works by scheduling and executing a predetermined event and callback function at a specified cycle. In essence, the discrete event simulation engine's scheduler will call the function passed to it when the number of cycles provided by the developer transpires. Ruby employs a slightly different technique. In Ruby messages are enqueued in buffers linking modeled system elements together. The buffers impose variable latency and bandwidth on inserted events. Simulation execution proceeds by invoking a callback function for the next scheduled event in a given event buffer.

In comparison to the modeling methodologies incorporated in the other computer architectural simulation systems presented here, KnightSim utilizes a different approach to event execution by implementing events as independently executable x86 "KnightSim Contexts". As presented in Sec. I, KnightSim Contexts encapsulate all of the functionality and interfaces associated with a single simulated system element in an executable package. KnightSim Contexts are treated as simulation objects that are scheduled for execution by an advance and await mechanism. In this approach occupancy and contention are then automatically modeled by KnightSim Contexts. In the other approaches discussed here, researchers must endeavor to carefully model the latency, occupancy, and contention incurred by the modeled resource. Since these simulation features are not an inherent part of the mechanism, such modeling must be implemented manually with a collection of events, flags, and appropriate execution timings.

Other parallel discrete event simulation techniques have previously been researched as well, such as distributed computing [12], [13], processor behavior prediction [14], [15], timing approximations [16], and instruction-driven timing [17]. In comparison, Parallel KnightSim parallelizes event execution by dividing KnightSim Contexts into batches for execution in parallel at each simulated cycle.

III. KnightSim Implementation Methodology

The makeup of a KnightSim Context is shown in Alg. 1. The context is defined by functions that encapsulate all of the user's desired functionality and interfaces associated with the simulated element. During simulation execution, contexts await until they are notified that they should execute by an advance from one or more previously running contexts, but will not execute until they are ready. Contexts currently in the run state may pause any number of simulation cycles or await a future event to assess a latency. Simulation cycle time increases once all contexts have either entered a pause or await state.

Algorithm 1 A KnightSim Context

```
procedure USER_FUNCTION(context* $ctx$)
    long long $i \leftarrow 1$;
    $\backslash\backslash Other\ local\ variables\ here$
    loop
        $await(my\_eventcount, i{+}{+}, ctx)$;
        $\backslash\backslash Do\ work\ after\ being\ advanced$
        $pause(1, ctx)$;                        ▷ Charge a latency
        $\backslash\backslash Finish\ doing\ work$
        $advance(neighboring\_eventcount, ctx)$;
        $\backslash\backslash Clean\ up\ and\ return\ to\ await\ state$
    end loop
    return
end procedure
```

The context's assessed latency during the paused or await state provides the mechanism to automatically model the occupancy of that context as no other work can be performed by the context during that time. Contention is automatically modeled as contexts must compete for modeled system resources. Individual contexts stall by pausing or awaiting as they wait for access to a particular resource. These additional stalls result in longer access latency for current and subsequent

Algorithm 2 Globals

```
globals
    if $wordsize == 64$ then
        typedef long int $\_jmp\_buf[8]$;
    else if $x86\_64$ then
        typedef long long int $\_jmp\_buf[8]$;
    else
        typedef int $\_jmp\_buf[6]$;
    end if

    typedef struct $context${
    _jmp_buf $buf$;                         ▷ Buffer for CPU registers
    unsigned long long $count$;
    void $(*start)$(struct $context*$);     ▷ Context's func
    char$*$ $stack$;                        ▷ Context's stack
    int $stacksize$;
    struct $context*$ $next\_ctx$;          ▷ Context's batch list
    } context;

    typedef struct $eventcount${
    struct $context*$ $next\_ctx$;
    unsigned long long $count$;
    } eventcount;
end globals
```

Algorithm 3 Context Initialization

```
procedure CTX_INIT($(*func)$(context$*$), int $size$)
    context$*$ $ctx \leftarrow$ NULL;
    $ctx \leftarrow$ (context$*$)malloc($sizeof$(context));
    $ctx{-}{>}count \leftarrow sim\_cycle$;
    $ctx{-}{>}stack \leftarrow$ (char$*$)malloc($size$);
    $ctx{-}{>}stacksize \leftarrow size$;       ▷ Stack overflow check
    $ctx{-}{>}start \leftarrow func$;           ▷ User defined function
    $ctx{-}{>}buf[ip] \leftarrow context\_start()$;
    $ctx{-}{>}buf[sp] \leftarrow stack\_top\_ptr$;
    $ctx{-}{>}next\_ctx \leftarrow$ NULL;
    $ctx\_hash\_insert(ctx, ctx{-}{>}count$&ROWS);
    return
end procedure
```

invocations as contexts wait for modeled hardware resources to become available.

A. Events as KnightSim Contexts

KnightSim implements events as KnightSim Contexts, which are independently executed by the CPU at runtime. A context is represented by a struct, that defines the context itself, and one or more eventcounts [18]. Pseudocode describing our implementation of contexts and eventcounts is shown in Alg. 2.

The context structure comprises a jump buffer, count, function pointer, stack pointer, stack size, and context pointer. The jump buffer is a primitive data type that is utilized by our hand implemented `setjmp()` and `longjmp()` assembly functions. Our implementations of `setjmp()` and `longjmp()` correct functional issues introduced by the recent additions of stack protection, pointer mangling, and source fortification in the Linux software stack. Usage of the standard Libc `setjmp()` and `longjmp()` functions will render this methodology non-functional in modern Linux distributions. Determination of the correct data type and size of the context's jump buffer is shown at the top of Alg. 2. 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 entry function. The stack pointer points to an allocated region of memory of user provided stack size. Each context's stack is unique, resides in user memory space, and contains that context's execution data. Contexts execute in a shared memory space and can operate on global C/C++ objects as well. The context pointer is used to form a singly linked list that comprises a batch of contexts that are ready to run at a given cycle. When a context enters the pause or await state the next context in the list is executed until the list is empty.

Eventcounts are objects that provide a mechanism with which to determine if a context should be placed in the run or await state. Eventcounts comprise a count that is used as an incrementer and a pointer to a context that is awaiting an advance of the eventcount. The eventcount's count records the number of times the eventcount has been advanced. Contexts await the advance of eventcounts and when the counts of both an eventcount and context are equal the awaiting context runs. Typically, each context will have at least one unique eventcount assigned to it, but this is not required.

Context batches are stored via a hash table and are formed as each context enters the pause state. Contexts are added to the table by hashing the context's designated future execution cycle with the global hash table's number of rows minus one. The global hash table's number of rows is set as a power of two and must be large enough to ensure that pausing contexts form batches of only one future execution cycle. We find that a hash table size of 512 is more than sufficient to meet this requirement. This is an optimized approach that maintains a high level of performance and doesn't require a modulus operation. Selecting the next context batch to run requires hashing the current global cycle count with the global hash table's number of rows minus one. A count of the number of unique context batches referenced by the hash table is kept. Simulation ends when the global hash table count is zero or the simulation reaches some desired end point.

B. Initialization

Prior to simulation execution each user created eventcount and context is initialized. Eventcount initialization is straightforward and comprises the allocation of the eventcount with the use of `malloc()`, the initialization of the eventcount's count to zero, and the initialization of the eventcount's context pointer to `NULL`. Context initialization is shown in Alg. 3 and comprises the allocation of the context itself with the use of `malloc()`, initialization 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 entry function, manipulation of the instruction and stack pointers in the context's jump buffer, and finally insertion of the context itself into the applicable context batch. The entry function embodies the functionality of the element this context will simulate, as shown in Alg. 1.

The context's jump buffer is uninitialized after being created. Thus, we assign a starting instruction pointer and stack pointer by hand to give the context our desired starting position and unique stack memory. This manual configuration of the context's jump buffer is what makes each context independently executable. We ignore other CPU registers at initialization because they will be obtained the first time `setjmp`() is called. Additionally, we push a pointer to the context onto the context's stack for retrieval later. This allows us to resolve information about the context after the context's initial jump.

Our pseudocode shows an instruction pointer assignment as the head of a `context_start`() function. On initial execution, each context will first jump to the head of this function and then retrieve the pointer to itself. We then call the context's `start`() function and pass the pointer to the context itself for future access. Program execution is now placed at the head of the user's provided entry function with resolution of the assigned context, see Alg. 1. Additionally, our pseudocode shows a stack pointer assignment as the top of the allocated stack which is calculated as shown in Equ. 1.

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

The assignment of the instruction and stack pointers to the context's jump buffer is architecture dependent. The instruction pointer and stack pointers are assigned to jump buffer positions *five* and *four* in the 32bit Linux x86 environment and are assigned to jump buffer positions *seven* and *six* in the 64bit Linux x86 environment.

C. KnightSim Context Scheduling

Pseudocode showing the mechanisms responsible for providing KnightSim Context scheduling is shown in Alg. 4 and 5.

Placing KnightSim in the simulation state simply requires obtaining a pointer to the first context in the initial context batch and performing a `longjmp`() to the context's starting position. Subsequently, each context resides in either an await, ready to run, or running state until the end of simulation. In the single threaded version of KnightSim only one context is ever in the running state at a time. A transition between these states is 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 use of the `advance`() function. By advancing an eventcount, the designated eventcount's count is incremented and the eventcount's context pointer is checked. If the counts of both the context and eventcount are equal the context is removed from the eventcount and inserted next into the current context batch as a context that should run this cycle.

After a context completes its tasks, the context then transitions to the await state by use of the `await`() function.

Algorithm 4 Context Scheduling

```
1:  procedure ADVANCE(eventcount* ec, context* ctx)
2:      ec−>count++;
3:      if ec−>next_ctx and ec−>next_ctx−>count == ec−>count then
4:          ec−>next_ctx−>next_ctx ← ctx−>next_ctx;
5:          ctx−>next_ctx ← ec−>next_ctx;
6:          ec−>next_ctx ← NULL;
7:      end if
8:      return
9:  end procedure
10:
11: procedure AWAIT(eventcount* ec, count value, context* ctx)
12:     if ec−>count >= value then
13:         return;
14:     end if
15:     ctx−>count ← value;
16:     ec−>next_ctx ← ctx;
17:     ctx ← ctx−>next_ctx;
18:     if !setjmp(ec−>next_ctx−>buf) then
19:         if ctx then
20:             longjmp(ctx−>buf);
21:         else
22:             sim_cycle++;
23:             longjmp(context_select());
24:         end if
25:     end if
26:     return
27: end procedure
28:
29: procedure PAUSE(count value, context* ctx)
30:     value ← value+sim_cycle;
31:     context* ctx_ptr ← ctx;
32:     ctx ← ctx−>next_ctx;
33:     ctx_hash_insert(ctx_ptr, value&ROWS);
34:     if !setjmp(ctx_ptr−>buf) then
35:         if ctx then
36:             longjmp(ctx−>buf);
37:         else
38:             sim_cycle++;
39:             longjmp(context_select());
40:         end if
41:     end if
42:     return
43: end procedure
```

The context will assign itself a count on which it will await, remove itself from the current context batch, assign itself to the designated eventcount's context pointer, and store the current position in its jump buffer. Simulation execution can then jump to the next context in the current context batch or, if this batch is finished, increment the global cycle count and select the next batch. A running context may also assess a latency with the use of the `pause`() function. Assessing a latency stops the

Algorithm 5 Context Scheduling Continued

```
1: procedure CONTEXT_SELECT($void$)
2:     context* $ctx\_ptr \leftarrow$ NULL;
3:     if $table\_count$ then
4:         do
5:             $ctx\_ptr \leftarrow table[sim\_cycle$&ROWS$]$;
6:         while $!ctx\_ptr$ and $sim\_cycle$++;
7:         $table[sim\_cycle$&ROWS$] \leftarrow$ NULL;
8:         $table\_count--$;
9:     else
10:        $sim\_end()$;
11:    end if
12:    return $ctx\_ptr->buf$;
13: end procedure
```

current context from running until a future global cycle count is reached, where the context will then automatically resume execution. The pausing context is removed from the current context batch and added to a context batch in the global hash table that is awaiting the same future global cycle count. If the addition to the global hash table results in a new context batch record the global hash table's count is incremented. Lastly, we store the current position in the pausing context's jump buffer. Simulation execution can then jump to the next context in the current context batch or, if this batch is finished, increment the global cycle count and select the next context batch.

The next context batch is selected with the `context_select()` function. We select the next context batch by iterating through the global hash table until we obtain a valid pointer to a batch of contexts. The global cycle count is incremented with each required iteration and reference of the hash table. Each removal of a context batch from the hash table results in a decrement of the global hash table's count. As mentioned before, simulation ends when the global hash table's count reaches zero.

IV. Parallel KnightSim Implementation Methodology

We developed KnightSim with an eye towards ultimately parallelizing it. Therefore, parallelizing KnightSim only requires a few changes which we highlight in this section. In general, the approach to parallelizing KnightSim is summarized best as splitting a given cycle's context batch into a balanced group of smaller context batches and then executing the group of context batches over an appropriate number of threads. This results in a discrete event driven simulation methodology that automatically parallelizes event execution at the cycle level.

Parallel KnightSim utilizes a pool of POSIX threads and a 2D global hash table, both of configurable size. Threads are assigned to columns in the global hash table based on their thread IDs. Pseudocode showing our thread control function is shown in Alg. 6. After creation, each thread sets its thread affinity and then spins while it waits for simulation execution to begin. Once simulation execution begins, each thread performs a `setjmp()` which records that specific position for

Algorithm 6 Thread Control

```
1: procedure THREAD_START(void* $id$)
2:     volatile bool $lflag \leftarrow false$;
3:     $thread\_set\_affinity((\textbf{long})id)$;
4:     while$(gflag != lflag)\{\}$;        ▷ Wait for sim execution
5:     $setjmp(thread\_buf[pthread\_self()\ mod$ SIZE$])$;
6:     $lflag \leftarrow !lflag$;        ▷ Invert the local status flag
7:     if $\_\_sync\_sub\_and\_fetch($&$threadnum, 1)$ then
8:         while$(gflag != lflag)\{\}$;        ▷ wait for last thread
9:     else
10:        $sim\_cycle$++;
11:        $threadnum \leftarrow$ SIZE;
12:        $gflag \leftarrow lflag$;        ▷ Last thread resets the flags
13:    end if
14:    $context\_select((\textbf{long})id)$; ▷ Contexts run next batch
15:    return
16: end procedure
```

each thread to return to at the end of each cycle. Threads access their jump buffers by use of a global jump buffer array and hash of their pthread handles and thread pool size. The threads then perform their first global cycle synchronization by entering a cycle barrier. Each thread inverts a local barrier flag and then atomically decrements a global int with a value of the thread pool size. All threads but the last to arrive spin while waiting for a global barrier flag update. The last thread to arrive is tasked with incrementing the global cycle count and resetting the global int. The last thread then inverts the global barrier flag as well, which releases all threads and places overall execution in the next cycle.

At the start of each new cycle all threads enter a modified `context_select()`. Like before, each thread checks the value of the global hash table's count. If the count is greater than zero each thread then consults the appropriate column and row of the context table and determines if there is a context batch to run this cycle or not. If a thread finds a context batch it removes the context batch from the global hash table and then atomically decrements the global hash table's count. The thread then performs a `longjmp()` to the last position of the first context in the batch. If a thread does not find a context batch to run the thread performs a `longjmp()` back to `thread_start()` where the thread then enters the cycle barrier and waits for the other running threads to return. When simulation ends, each thread returns and is collected by the main process with use of `pthread_join()`.

Simulation flow control in `await()` and `pause()` is modified as well. At the end of `await()` and `pause()` each thread performs a `longjmp()` to the next context's last position if there is another context to run. However, if there are no longer any contexts to run the thread's work has ended for the cycle. The thread then performs a `longjmp()` back to `thread_start()` where the thread then enters the cycle barrier. All threads return to `thread_start()` and enter the cycle barrier at the end of each cycle before simulation execution continues into the next cycle.

V. Performance Evaluation

For our performance evaluations, we make direct performance comparisons between KnightSim, Parallel KnightSim, and the discrete event driven simulation engines found in Gem5 [3], Multi2Sim [4], and M2S-CGM [2]. We make comparisons to this selection of discrete event driven simulation engines because the simulators in which they are used are relevant, widely recognized, and have been used in recent computer architectural simulation related publications. For the purposes of our experiments, we refer to the discrete event driven simulation engines found in Gem5, Multi2Sim, and M2S-CGM as Gem5-Event, Esim, and The Threads Package respectively. We refer to KnightSim and Parallel KnightSim as KS and PKS_N respectively. For PKS, the "_N" denotes the number of specified threads used in each PKS trial.

A. Experimental Setup

We conduct all experiments on our test system comprising a 16 core Intel Xeon E5-2697A v4 processor running at 2.6 GHz - 3.6 GHz and ample system memory running at 2400 MHz. In all test cases we measure execution time over the equivalent `simulate()` function. Our measured execution times do not include time spent in regions of code associated with setup, initialization, or cleanup activities. Additionally, we have removed non-essential code, like asserts, from each test application.

Gem5-Event and Esim employ a similar implementation approach that establishes an event list with associated callback functions upon initialization. During execution, events are scheduled to run in either the current cycle or a future cycle using an equivalent `schedule_event()` function. Scheduled events are placed in a data structure and removed for execution at a later simulation cycle. Gem5-Event declares class objects as sim objects whose member functions can be declared as events. Therefore, we implement an event in Gem5-Event as a single class member function that is initially scheduled to run by the class's constructor during initialization time. Esim declares domain event handlers that are meant to handle a number of domain specific sub events. Thus, we implement an event in Esim as a single domain level event that is registered and scheduled to run at initialization time. For both Gem5-Event and Esim, each time an event is executed the event schedules itself to run again in one cycle.

KS, PKS, and The Threads Package implement events as contexts, however the implementation of The Threads Package is completely different and does not benefit from the extensions presented in this paper. As discussed in Sec. III-C, scheduling a pause of one cycle during context execution is functionally identical to scheduling an event to occur one cycle later, as in Gem5-Event and Esim. For KS, PKS, and The Threads Package an event is implemented as a single context that is registered and scheduled to run at initialization time. Each time the context is run, the context schedules itself to run again after one cycle by pausing one cycle.

B. Determining Event Engine Usage

We wish to determine how often a typical computer architectural simulator makes use of its event engine. To answer this question we need to determine what a realistic range is in terms of (1) the average and maximum number of executed events per simulated cycle and (2) the average and maximum number of physical cycles per event. These two measurements give us a sense of the amount of pressure placed on the event engine and how many physical cycles it takes to process an event on average. To measure these values we utilize M2S-CGM and run a sampling of the Rodinia OMP benchmarks [19]. We take measurements over the benchmark's parallel section while varying the size of the simulated system. M2S-CGM provides a system wide model with a configurable number of CPU cores, L1, L2, and L3 caches, switching network, system agent, memory controller, and SDRAM.

Sim Cores	**Avg Events**	**Max Events**
1	13	26
2	20	38
4	34	58
8	60	95
16	113	165
32	215	291
64	416	529
128	813	980

Physical Cycles	**Avg Cycles**	**Max Cycles**
	2680	9945

Table 1: Measured Event Engine Usage

Our findings are shown in Table 1 and are used to form the basis of our experiments. We establish that a realistic range of expected events per simulated cycle is approximately 13, or fewer, for small simulation models to approximately 980, or more, for large simulation models. From the results, it is apparent that the predominance of computer architectural simulation models used in relevant research would fall in the category of approximately 113 events, or fewer, per simulated cycle.

We also establish that a typical computer architectural simulator utilizing a context based event engine performs an average of 2680 physical cycles of work per event. However, we observed that in some cases this can be considerably higher. We utilize this data point to determine if the usage of Parallel KnightSim for the purposes of parallelizing computer architectural simulations is viable or not.

C. Experimental Results: Single Threaded

For our single threaded experiments we measure overall speedup over eight test cases where we execute 16, 32, 64, 128, 256, 512, 768, and 1024 events per cycle for one million cycles. The selected test cases provide a good range in terms of varied simulated system size, as established in Sec. V-B. The selected test cases also facilitate our comparisons to Parallel KnightSim executions in the following section. Prior to taking performance measurements we verified that all test applications function correctly by observing the value of a global int that was incremented by each event execution. However, in our measured experiment all events perform no work. We found one million simulated cycles to be more than sufficient to reach a steady state for performance measurements.

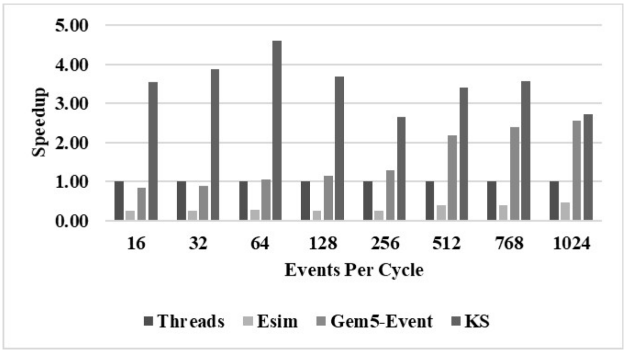

Figure 1: Single Threaded Performance Results

The experimental results for our single threaded executions are shown in Fig. 1. The results for each test case are normalized to the execution results of The Threads Package. KS demonstrated strong overall performance with an average speedup of 3.51, 11.95, and 2.8 over the execution results of The Threads Package, Esim, and Gem5-Event respectively. However, more importantly KS showed an average speedup of 4 over both The Threads Package and Gem5-Event in the range of 16 to 128 events per cycle. This range represents the most likely usage range of event engines in computer architectural simulations.

The results for KS show that our extensions to The Threads Package have provided a significant boost in the methodology's overall performance and that the methodology also outperforms those of Gem5-Event and Esim. Gem5-Event proved to scale very well with larger simulated system sizes, but did not outperform KS over the selected test case range. Esim provided a consistent performance baseline, but did not exceed the performance of KS, Gem5-Event, and The Threads Package. An inspection of Esim's code revealed that Esim performs a `calloc`() with the scheduling of each event and a `free`() at the end of each event's execution. This is the leading cause of the performance disparity between Esim and the other event engines.

D. Experimental Results: Multithreaded

Drawing on the experiences gained by implementing Parallel KnightSim, it is apparent that the two most critical factors impacting overall parallel performance in computer architectural simulations are (1) the number of events executed per cycle and (2) the amount of work each event performs when executing. These two factors inversely effect the negative impacts to parallel performance imposed by the global cycle barrier and other pthread related overhead. Parallel speedups are achievable once the combined effects of these two factors amortize the cost of the serial section.

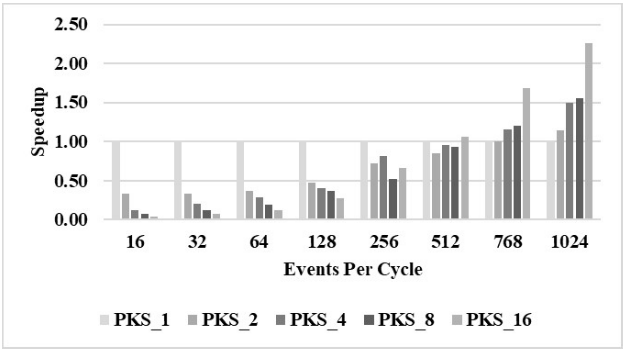

Figure 2: Multithreaded Performance Results Without Work

Experimental results regarding the impact of the number of events per cycle on overall parallel performance is shown in Fig. 2. In this experiment we maintain the configuration described in Sec. V-C, but parallelize the workload with PKS. When events perform no work the cost of the global cycle barrier and other pthread related overhead is apparent in the results. However, increasing the number of events per cycle begins to amortize the cost of the serial section when executing approximately 256 events per simulated cycle in parallel. At approximately 512 events per simulated cycle, and higher, parallelization results in measurable speedups.

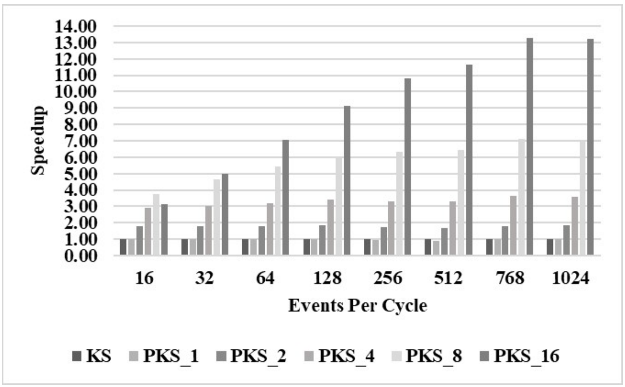

Figure 3: Multithreaded Performance Results With Work

We can determine the amount of work each event must perform to amortize the cost of the serial section by adding work to the events executed by both KS and PKS_1 until their overall performance is equal. The results of our trials showed that, on average, events must work for approximately 1700 physical cycles, in conjunction with the number of events being executed per cycle, to amortize the cost of the serial section. Fig. 3 shows the impact of the two factors combined. Again, we maintain the configuration described in Sec. V-C, but include a workload of approximately 1700 cycles in both KS and PKS events and normalize all results to those of KS. The results show that for each test case the cost of the serial section is amortized as both KS and PKS_1 have an equal overall performance of 1. We observed that PKS_2, PKS_4, PKS_8, and PKS_16 demonstrated an average speedup over KS and PKS_1 of 1.78, 3.30, 5.84, and 9.16, respectively.

Considering that PKS does not need to execute an unrealistically high number of events per cycle to gain parallel performance and that the imposed 1700 physical cycle threshold is lower than the number of physical cycles measured in Table 1, it is evident that utilizing PKS to parallelize small to large computer architectural simulations is viable and can result in measurable speedups over the established performance of KS.

E. Parallel KnightSim In The Wild

As discussed in Sec. IV, Parallel KnightSim automatically parallelizes event execution at the cycle level. However, thread safety must be taken into account when utilizing Parallel KnightSim. Race conditions can occur between contexts that share data and that are being executed by two or more threads in a given cycle. However, these race conditions can be avoided by assigning contexts that share data to the same context batch so that they are run by the same thread. In places where a natural division of the contexts is not possible the inclusion of thread-safe techniques in only those contexts will eliminate the race condition. Context stealing can also occur when a context of one thread advances the context of another thread. Context stealing can be avoided by using a "safety" context which allows the advancee's owning thread to complete the advancement task on behalf of the advancer's owning thread. Ultimately, optimized parallel simulation performance is gained by balancing the simulation model's size and simulated architectural structure with a properly specified number of threads and context-to-thread assignment.

VI. Conclusion

In the first half of this paper we introduce KnightSim and discuss the benefits of KnightSim's event implementation approach regarding how KnightSim Contexts automatically model simulated occupancy and contention. We then provide detail regarding KnightSim's implementation methodology. We discuss critical items pertaining to how KnightSim is used and how events are instantiated as KnightSim contexts. Then, we discuss the implementation methodology of Parallel KnightSim.

In the second half of this paper we report the results of a detailed performance analysis of discrete event driven simulation engines in computer architectural simulators and the results of a direct comparison between KnightSim, Parallel KnightSim, and the discrete event driven simulation engines found in Gem5, Esim, and M2S-CGM. Our study provides insight into the average number of events executed per simulated cycle and average number of physical cycles it takes to process an event in a typical computer architectural simulator. We establish that small simulation models execute approximately 13 events per simulated cycle, or fewer, and that large simulation models execute approximately 980 events per cycle, or more. Our overall performance results showed that on average KnightSim achieves speedups of 2.8 to 11.9 over the other discrete event driven simulation engines presented in this paper. Our results also show that, on average, Parallel KnightSim can achieve speedups over KnightSim of 1.78, 3.30, 5.84, and 9.16 in 2, 4, 8, and 16 threaded executions respectively. Based on the performance results presented here and on the additional benefits of KnightSim's context-based approach we believe that KnightSim is a promising tool for use in the development of future computer architectural simulations.

Ready-made and fully working implementations of KnightSim and Parallel KnightSim, with usage examples, are made available as free software and can be found on GitHub.

References

[1] 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.

[2] C. E. Giles and M. A. Heinrich, "M2S-CGM: A detailed architectural simulator for coherent cpu-gpu systems," in *2017 IEEE International Conference on Computer Design (ICCD)*, Nov 2017, pp. 477–484.

[3] 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.

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

[5] M. Chaudhuri, M. Heinrich, C. Holt, J. P. Singh, E. Rothberg, and J. Hennessy, "Latency, occupancy, and bandwidth in dsm multiprocessors: a performance evaluation," *IEEE Transactions on Computers*, vol. 52, no. 7, pp. 862–880, Jul 2003.

[6] 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.

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

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

[9] 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.

[10] 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.

[11] 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.

[12] K. M. Chandy and J. Misra, "Distributed simulation: A case study in design and verification of distributed programs," *IEEE Transactions on software engineering*, no. 5, pp. 440–452, 1979.

[13] S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, and D. A. Wood, *The Wisconsin Wind Tunnel: virtual prototyping of parallel computers*. ACM, 1993, vol. 21, no. 1.

[14] D. R. Jefferson, "Virtual time," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 7, no. 3, pp. 404–425, 1985.

[15] D. M. Nicol, *Parallel discrete-event simulation of FCFS stochastic queueing networks*. ACM, 1988, vol. 23, no. 9.

[16] P. Ren, M. Lis, M. H. Cho, K. S. Shim, C. W. Fletcher, O. Khan, N. Zheng, and S. Devadas, "Hornet: A cycle-level multicore simulator," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 6, pp. 890–903, 2012.

[17] D. Sanchez and C. Kozyrakis, "Zsim: fast and accurate microarchitectural simulation of thousand-core systems," in *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3. ACM, 2013, pp. 475–486.

[18] 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

[19] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, Oct 2009, pp. 44–54.