Sitar: A Cycle-based Discrete-Event Simulation Framework for Architecture Exploration

Neha Karanjkar[1] [a] and Madhav Desai[2]
[1]Indian Institute of Technology Goa, India
[2]Indian Institute of Technology Bombay, India

Keywords: Simulation Framework, Discrete-Event, Cycle-based, Open-source, Parallel, C++.

Abstract: Sitar is an open-source framework for modeling discrete-event, discrete-time systems. It consists of a modeling language and a lightweight simulation kernel. Sitar is specifically targeted for architecture-level modeling and fast simulation of computer systems, though it can be used for other kinds of discrete-time systems as-well. The modeling language allows the description of a system's structure as an interconnection of hierarchical, concurrent entities. The behavior of each entity can be described in an imperative manner using constructs such as time-delays, conditional `wait` statements, fork-join concurrency and loops. C++ code can be embedded directly into the description in a well-defined manner, allowing the modeler to use the flexibility and object-oriented features of C++. A model written in this language gets translated to C++ code, which can in-turn be compiled with the simulation kernel to obtain a single simulation executable, or can be linked with external libraries for co-simulation. The simulation kernel uses a two-phase, cycle-based execution algorithm, and has been parallelized using OpenMP for fast and scalable simulation on modern multi-core systems. The framework provides several features to ease the modeling effort, such as in-built logging, syntax highlighting and systematic error reporting for the Sitar language. In this paper, we describe the design and architecture of Sitar, and briefly discuss our experience with its use for multi-core design exploration studies.

1 INTRODUCTION

Simulation plays a key role in system-level design exploration and optimization of modern computer systems. The systems to be modeled are often very large and complex, and therefore initial design exploration makes use of models written at a high level of abstraction. Creating such models necessitates frameworks that can support fast, efficient and scalable simulations, while also being expressive enough to ease modeling and debugging effort.

Most simulators used in computer architecture research provide the user parametrized models of components such as processors, caches, and interconnects built directly using a high-level programming language such as C/C++ or Java (Akram and Sawalha, 2019). Although these programming languages allow modular descriptions, they have no built-in features for describing system structure (as an interconnection of concurrent entities) or the behavior of entities with respect to simulation time. Therefore such simulators bundle together the model description along with custom code for advancement of simulation time.

When the focus is on creating new models rather than using pre-existing configurable models provided by simulators, discrete-event modeling frameworks may be used. Discrete-event frameworks can be broadly divided into *Event-Driven* and *Cycle-based* (also known as Time-stepped) based on the simulation approach. An event-driven approach requires the use of a global *Event Queue* (often implemented as a Priority queue data structure) for keeping track of dynamically scheduled events, sorted by their time-stamp. Time advancement is performed by directly updating the simulation time variable to the time-stamp of the earliest scheduled, unprocessed event in the queue. All events scheduled at this time are processed one-by-one. Actions or state-updates to be triggered by the occurrence of an event are executed using callbacks. On the other hand, a cycle-based approach assumes that all activities in the system occur only at integer multiples of some fixed time unit. Time advancement occurs in fixed-size steps. At each time-step, active entities/processes may be executed

[a] https://orcid.org/0000-0003-3111-1435

Karanjkar, N. and Desai, M.
Sitar: A Cycle-based Discrete-Event Simulation Framework for Architecture Exploration.
DOI: 10.5220/0011320000003274
In *Proceedings of the 12th International Conference on Simulation and Modeling Methodologies, Technologies and Applications (SIMULTECH 2022)*, pages 142-150
ISBN: 978-989-758-578-4; ISSN: 2184-2841

to update model state. Cycle-based simulation is particularly suited to systems that naturally have a fixed time-step (such as clocked digital systems, discrete-time queues and computer networks).

An event-driven approach may be more efficient for simulating systems where the frequency of activities vary widely over time or across components in the system. However it is fundamentally difficult to parallelize the event-driven simulation algorithm, although there exists a large body of research in this direction (Fujimoto, 1990). This is because the event queue either needs to be shared across multiple parallel processes/threads, or divided in a coherent manner. The difficulty also arises because the time increments are dynamically computed and are not known up-front.

VHDL, Verilog and SystemC (OSCI, 2021) are some examples of discrete-event frameworks that allow modeling of system structure and behavior of concurrent entities with respect to simulation time. While VHDL and Verilog are meant for detailed hardware-level modeling, SystemC has been used for hardware-level, Register-Transfer-Level(RTL) as-well-as system-level simulations (Fummi et al., 2008) and is designed to support incremental transition from abstract descriptions to detailed hardware-level models.

For building purely architecture-level models of large clocked systems for rapid design exploration, SystemC may not be the best choice. This is because SystemC uses an event-driven simulation approach which is fundamentally difficult to parallelize. SystemC was intended to also support hardware-level modeling and provides constructs such as *signals*. (A signal is essentially a state variable which when updated, can automatically trigger activity in other components that are sensitive to the signal.) Although parallelization of SystemC simulations using parallel discrete-event (PDES) algorithms has been explored (Dömer et al., 2012) the achievable speedup is low for most benchmarks. General-purpose discrete-event simulation frameworks such as SimPY (SimPY, 2021) typically do not provide any special constructs for modeling system structure and use event-driven approach since their primary focus is on modeling flexibility. A cycle-based simulation approach can be easier to parallelize and is often well-suited for architecture-level modeling of clocked-systems. Cycle-based simulation has been used in libraries such as Cascade (Grossman et al., 2013) and SystemCASS(Buchmann and Greiner, 2007), which is a cycle-accurate variant of the SystemC kernel.

This paper describes **Sitar** - a framework for modeling discrete-event, discrete-time systems. It consists of a modeling language and a cycle-based simulation kernel. The design of Sitar is driven by the goal of supporting fast, scalable and parallel simulation, while also being expressive enough to ease modeling and validation effort. Its key design feature is a **two-phase, cycle-based simulation algorithm** which makes the simulation efficient and easy to parallelize.

Sitar has been in development since 2013. It was initially developed as an internal tool, specifically targeted for creating architecture-level models of Multi-core systems for design exploration. Subsequently the design and simulation kernel were improved and simplified, retaining the most useful features to create Sitar Version 2.0 which is available as open-source under an MIT licence. The online repository is present at the following url: `https://nehakaranjkar.github.io/sitar/`.

The main focus of this paper is on presenting the design and architecture of Sitar, along with its rationale. We first summarize the differentiating aspects of Sitar in the following paragraphs. In Section 2, we present an overview of Sitar, describe its underlying execution model and discuss the parallel simulation strategy. In Section 3 we present an overview of the Sitar modeling language, using illustrative examples. The framework has been used for creating a detailed cycle-accurate model of a multi-core system for design exploration studies. In Section 4 we describe our experience and the performance and scalability observed for this use case. We present a summary and discuss future work and conclusions in Section 5.

1.1 Related Work

The differentiating aspects of Sitar in the context of other cycle-based simulation frameworks are summarized below:

- Sitar uses a *two-phase* cycle-based execution model (described in detail in the next section) which makes it possible to execute components of the model in parallel, in a deterministic way.

  Parallel execution is non-trivial because the exact order in which the individual components in the model are executed can affect the results, leading to a race condition. In frameworks such as Cascade (Grossman et al., 2013) and SystemCASS (Buchmann and Greiner, 2007), deterministic execution is achieved by first building a static dependency graph between the components. The behavior of components in each cycle is executed in this statically-computed order. If there are loops in this dependency graph, the components within the loop are executed multiple times until convergence.

This is not the approach used in Sitar. In Sitar, each cycle is divided into two *phases*. Individual components (modules) are allowed to perform input and state-update actions in the first phase, and output actions exclusively in the second phase. Parallel execution is performed simply by mapping these components to separate threads, which synchronize at the end of every phase. Consequently, the order of execution among individual threads does not affect the results. A similar two-phase approach has been reported in the Hornet Multi-core simulator (Ren et al., 2012). However, Hornet is not a modeling framework. Rather, it is a specific configurable model of a multicore system. Sitar is a framework and a language for modeling cycle-based systems and can be used for modeling any kind of discrete-time system.

- Another important aspect that differentiates Sitar from frameworks such as SystemC is that Sitar provides a rich, custom modeling language. Descriptions in this language can be translated automatically to more verbose C++ code. In contrast, SystemC is a C++ library. Constructs such as time-delays are to be expressed in SystemC using *macros*. This often leads to verbose model descriptions that may be difficult to maintain or debug. The utility and conciseness of the modeling language is illustrated by examples in Section 3.

2 ARCHITECTURE

2.1 Overview

The Sitar framework consists of a modeling language and a lightweight, cycle-based simulation kernel. A system in Sitar can be described as a set of *modules* (which are behavioral entities or active processes in the system) communicating over channels called *nets*. All modules are assumed to run concurrently on a global clock. The language provides a means for describing the system structure (as an interconnection of modules) in a hierarchical and modular way. The behavior of each module can be described in an imperative manner as a *sequence* of statements. The statements include conditional and unconditional time-delays, branch and loop constructs, parallel blocks and instantaneous code blocks. In addition, C++ code can be embedded into a module description in a straightforward and well-defined manner. Descriptions written in this language get translated to C++ code and can be compiled to get a simulation executable. Listing 1 shows a simple example and its

```
//A hello−world example
module Top
  behavior
    $ log<<endl<<"Hello World!"; $;
    wait(2,0); //wait 2 clock cycles
    $ log<<endl<<"Hello again!"; $;
    wait(3,0); //wait 3 clock cycles
    $ log<<endl<<"Bye!"; $;
    stop simulation;
  end behavior
end module
//Output:
//
// (0,0)TOP      :Hello World!
// (2,0)TOP      :Hello again!
// (5,0)TOP      :Bye!
// simulation stopped at time (5,0)
```

Listing 1: A Hello-World Example.

simulation output. The Sitar language parser/translator has been built using Antlr V3 (Parr, 2009). It translates each module description into a C++ class. The generated classes inherit methods from the simulation kernel to allow execution of their behavior. They can be compiled along with the simulation kernel code to obtain a single simulation executable, *or* linked to external models (such as processor front-ends) for co-simulation using direct function calls or through inter-process communication.

The Sitar framework includes systematic support for logging. The logging mechanism is implemented using C++'s `std::ostream` library. Individual modules can send their logs to a common stream or separate streams. A module instance's hierarchical name and the time-stamp are prefixed automatically to each log message. The logging can be enabled/disabled at compile-time or run-time, and the framework also provides fine-grained control to enable/disable logging for specific modules during simulation. This feature is quite useful when debugging very large systems or long simulations. By enabling logging for only specific modules in the system under specific conditions, it is possible to generate a focussed activity trace without having to change the model code.

The simulation kernel is cycle-based and uses a **two-phase execution algorithm** for deterministic simulation. It has been parallelized using OpenMP for fast simulation on shared-memory multi-core systems. Individual modules or a group of modules can be mapped to separate OpenMP threads that synchronize at fixed time-steps. The execution model and parallelization are described in detail in the following paragraphs.

2.2 Execution Model

The basic components in a Sitar system are *modules* and *nets*. Modules are behavioral entities in the system and nets are channels of communication between them. All modules run concurrently on a single clock. A module can communicate with another module via transfer of *data tokens* (which are packets of information) over a net connecting the two modules. Each net provides a fixed amount of FIFO buffering for data tokens. Nets are passive components, and their state can change only upon input or output actions by modules. A module's interface to a net is called a *port*. A port can either be an *inport* or an *outport*. Each net is connected to exactly one inport and one outport. Modules can be hierarchical, containing instances of other modules. The system description must contain a single module named `Top` which represents the top-level module in the hierarchy to be instantiated for simulation. Modules can optionally have a behavior block which describes the behavior of the model over time. Figure 1 illustrates a system with four modules contained inside the Top module.

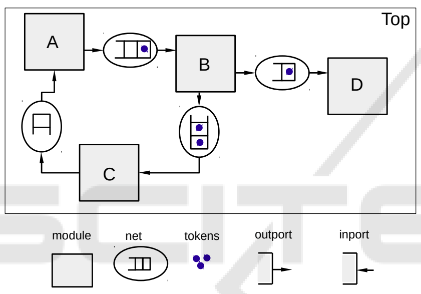

Figure 1: Example of a system in Sitar.

Simulation is cycle-based and uses a **two-phase execution algorithm**. In this algorithm, each clock cycle is divided into two phases : *phase 0* and *phase 1*. A module is allowed to input tokens from a net in phase 0 only, and output tokens to a net in phase 1 only. Thus the state-updates to a net occur in a race free (deterministic) manner. This restriction leads to a simple simulation algorithm that is easy to parallelize. In this simulation algorithm, in each phase, the behavior of each module in the system is executed exactly once, as follows:

```
cycle = 0
while (cycle < total_simulation_cycles)
{
   phase=0
   for each module m :
       m.execute_behavior(cycle,phase)
   phase=1
   for each module m :
        m.execute_behavior(cycle,phase)
   cycle=cycle+1
}
```

The two phase approach avoids race conditions in simulation as the result does not depend on the order of execution among modules. This is the key for enabling straightforward parallelization. To perform parallel simulation, the set of modules may be divided and mapped to different threads running concurrently and synchronizing at the end of each phase. It is important that between any pair of modules that are mapped to separate threads, the communication must occur solely via nets, and not through other shared variables in an ad-hoc manner. Updating the state of a net by input/output from modules need not be placed within a critical section, as the algorithm guarantees that at-most one module will update the net in a single phase.

As a consequence of the two-phase execution, the propagation of information from one module to another over a net incurs **a delay of at-least one clock cycle**. This is similar to the communicating Moore-machines paradigm. If there are cycles in the structure (for example, the loop via modules A, B and C in Figure 1), and if the restriction of 1-cycle delay is not applied, events can propagate from input of a module, to its output, and back to its input instantaneously, requiring multiple executions of module behavior within a cycle until convergence, or necessitating building a dependency graph and executing module behavior in the order of dependency. (The later approach is in-fact used by the Cascade framework (Grossman et al., 2013).) The two-phase restriction imposed by Sitar leads to an efficient, easy-to-parallelize simulation.

Instantaneous communication or instantaneous dependency loops between concurrent components in the system can be modeled by placing these concurrent components within a single module as parallel branches of a fork-join construct. The modeling language supports this via `procedures` and `parallel` constructs as described in Section 3 (see Listings 4 and 5 as examples). These concurrent branches can be executed multiple times until convergence within a single module. The execution order of individual parallel components within a module's behavior block is fixed (determined by the order of component declarations in the model), and thus cannot lead to race conditions.

The simulation kernel has been parallelized using OpenMP. The division of the set of modules and their mapping to OpenMP simulation threads can be dynamically determined using OpenMP's default scheduler or statically specified by the modeler for a balanced work-division.

3 MODELING LANGUAGE

In this section, we present an overview of the Sitar modeling language using illustrative examples. The language is case-sensitive, and supports embedding of C++ code at several places in a well-defined manner. The basic design units in a sitar description are *modules* and *procedures*. Modules are the basic structural entities that can be instantiated independently. A module can be hierarchical (containing instances of other modules) and can optionally also have a *behavior block*, describing its behavior with time as a sequence of statements. A procedure describes a sequence of actions that can be invoked from within the behavior block of a module or from within another procedure (nesting of procedures is allowed). This construct is meant for making behavior descriptions modular and reusable. A design unit can be defined once and instantiated multiple times.

3.1 Describing System Structure

The modeling language supports description of system structure (interconnection and hierarchy of modules) and attributes such as capacities of nets and widths of ports (to model communication bandwidth). A module can contain instances of other modules, declared using the `submodule` keyword. Module descriptions can be parameterized, and multiple instances can be created with different parameter values. Generation of regular structures such as arrays of modules and nets and their connections in a regular pattern is supported via `generate` constructs. This is similar to, and inspired from the `generics` and `generate` constructs in VHDL language. These aspects are illustrated by an example of a Shift-Register model in Listing 2 where the number of stages and delay of each stage are parameterized.

3.2 Describing Module Behavior

The behavior of a module can be described within a `behavior` block as a *sequence* or statements separated by semicolons. The statements execute one by one and can be of two types:

- **Atomic Statements:** These can be C++ code snippets, `wait(duration)`, conditional `wait until (condition)` or simulation control statements that are used for stopping the simulation of a particular module or the entire system upon certain conditions.
- **Compound Statements:** are statements that can in-turn contain a nested sequence of any statements. Examples of compound statements are `do-while` loops, `if-else` statement, `parallel` blocks and `procedures`.

```
// The system consists of a Producer and a Consumer
// connected via a ShiftRegister. The number of stages and
// delay of each stage of the ShiftRegister are parameters.

module Top
  //Instantiate a shift register
  //with 3 stages, and a per−stage delay of 1
  submodule S : ShiftRegister<3,1>

  //Instantiate the Producer and Consumer
  submodule P : Producer
  submodule C : Consumer

  //Connect the Producer and Consumer
  //to the two ends of the ShiftRegister
  P.out_port => S.n[0]
  C.in_port  <= S.n[3]
end module

module ShiftRegister
  //declare module parameters
  //and their default values
  parameter int N = 1     //number of stages
  parameter int DELAY = 1 //delay of each stage


  //The ShiftRegister consists of N stages
  submodule_array stage[N]  : Stage<DELAY>
  //N+1 nets to connect all stages
  net_array n[N+1] : capacity 1

  //connect the stages via nets
  for i in 0 to (N − 1)
    stage[i].in_port  <= n[i]
    stage[i].out_port => n[i+1]
  end for
end module

module Stage
  parameter int DELAY = 1
  inport in_port
  outport out_port
behavior
  // describe the behavior of each stage
  //...
end behavior
end module
```

Listing 2: A Shift Register model in Sitar.

Listing 3 shows examples of some of these statements used inside a behavior block. **C++ code snippets** delimited by dollar `$` characters can be used for embedding C++ code into a description. The embedded code gets pasted as-is in the generated C++ classes at various locations. Inside a behavior block, it can be used as an atomic statement or an expression. Code snippets prefixed by `incl`, `decl` and `init` can be used to add code to the header section, declarations region in the class body or the class constructor of a module respectively.

Parallel Block Statements: are used for modeling fork-join concurrency within a module. A parallel block is a compound statement containing nested sequences (separated by `||` symbols) that are concur-

```
//An example showing atomic and compound statements
module Top
 behavior
   //Declare a variable. This becomes a data member
   decl $int x;$;

   //This goes into the constructor
   init $x=0;$;

   //An atomic statement to read user−input
   $ std::cout<<"\nEnter a number:";
       std::cin>>x;$;

   //An if−else statement containing
   //nested do−while and other statements
   if($x%2==0$) then
     //Enter this branch if x is even
     //Print the time in every phase
     //until time exceeds 3.
     do
        wait(0,1); //wait for one phase
        $std::cout<<"\nTIME="<<current_time;$;
     while($current_time<=time(3,0)$) end do;
     stop simulation;
   else
     //Enter this branch if x is odd
     //stop simulation after 2 cycles
     wait (2,0);
     stop simulation;
   end if;
 end behavior
end module
// Output:
// Enter a number:4
//
// TIME=(0,1)
// TIME=(1,0)
// TIME=(1,1)
// TIME=(2,0)
// TIME=(2,1)
// TIME=(3,0)
// TIME=(3,1)
// simulation stopped at time (3,1)
```

Listing 3: An example showing Atomic and Compound statements in Sitar.

rently active. Listing 4 shows an example and its output. As shown in this example, there can be dependencies between the branches of a parallel block. In the implementation, each branch may be executed multiple times in a single phase until convergence. The entire parallel block terminates once all of the constituent sequences terminate or until a `stop` statement is encountered. **Procedures** are a useful feature in Sitar meant for modular description. A Procedure encapsulate a sequence of statements that need to be invoked multiple times or at multiple locations in a description. They can have local variables and parameters, similar to a module. A procedure definition also gets translated to C++ class. An instance of this class becomes a data member of the module invoking it. An example is shown in Listing 5. Other features such as the use of data-tokens, logging control and simulation control statements are described with examples in the user manual available inside the

```
//An example showing Parallel block statement

module Top
   behavior
     decl $bool x;$;
     init $x=false;$;
     wait(1,0);
     [
       //This is the first branch
       wait (2,0);
       do
          $x=true;$; //Set x
          $log<<endl<<"In branch A, set x";$;
          wait until($x==false$);
       while(1) end do;
     ||
       //This is the second branch
       do
          wait until($x==true$);
          wait(1,0);
          $x=false;$; //Reset x
          $log<<endl<<"In branch B, reset x";$;
       while(1) end do;
     ||
       //This is the third branch
       wait(4,0);
       stop simulation;
     ];
   end behavior
end module
// Output:
//
// (3,0)TOP      :In branch A, set x
// (4,0)TOP      :In branch B, reset x
// (4,0)TOP      :In branch A, set x
// (5,0)TOP      :In branch B, reset x
// (5,0)TOP      :In branch A, set x
// simulation stopped at time (5,0)
```

Listing 4: A Parallel block statement.

online repository (Karanjkar and Desai, 2022). The repository also includes a syntax specification for the language and Syntax highlighting support for the Vim text editor tool.

4 USE-CASE

In this section we discuss a particular use-case of Sitar and describe the modeling experience and simulation performance/speedup obtained.

System Model: The Sitar framework was used for creating a cycle-accurate, parameterized model of a Multi-core system for design exploration studies. The system is illustrated in Figure 2. It consists of *m* processors, with *n* cores per-processor (where *m* and *n* are model parameters). The processors are connected to *m* memory modules, forming an *m*-way Non-Uniform Memory Access (NUMA) configuration. Each core implements the Sparc V8 instruction set. The cache subsystem comprises per-core split L1 I/D caches, a per-core unified L2 cache, and a shared L3 cache. Coherency is maintained using a hierarchical directory-

```
//Definition of a procedure
//that performs some action periodically
procedure MyProcedure
 parameter int PERIOD=1
 behavior
 do
  wait(PERIOD,0);
  $cout<<"\nTIME="<<current_time;$;
  $cout<<"in Procedure "<<instanceId();$;
 while(1) end do;
 end behavior
end procedure


module Top
 //Create two instances of the
 //Procedure with different periods
 procedure p1 : MyProcedure<1>
 procedure p2 : MyProcedure<2>
 behavior
  //run p1 and p2 in parallel,
  //stop simulation after 5 cycles.
  [run p1 || run p2 || wait(5,0); stop simulation;]
 end behavior
end module

// Output:
// TIME=(1,0)in Procedure p1
// TIME=(2,0)in Procedure p1
// TIME=(2,0)in Procedure p2
// TIME=(3,0)in Procedure p1
// TIME=(4,0)in Procedure p1
// TIME=(4,0)in Procedure p2
// TIME=(5,0)in Procedure p1
// simulation stopped at time (5,0)
```

Listing 5: A Procedure block.

based MESI protocol. Interconnect between successive levels in the memory hierarchy is a full-crossbar (labeled `X` in Figure 2) with parametrized link delays.

Architectural components in the memory subsystem (such as caches, memory modules and interconnect switches) are modeled at the functional as well as cycle-accurate timing levels. Thus, the timing of loads/stores and coherence requests flowing through the memory subsystem is modeled in detail. The cores are in-order and all instructions other than loads/stores are assumed to execute in one cycle. The model can run binaries compiled for Sparc V8. In the implementation, the purely functional aspects of the model (such as the instruction decoder, the coherence protocol etc) are described as C++ classes. This code is imported into the Sitar module description of components such as cores and caches to create a cycle-accurate model. The modeling language features such as `generate` statements, `parameters` and `procedures` were found to be highly useful in making the description modular and configurable. The number of cores, cache configurations and the delay parameters in every module can be configured at the time of instantiation. Each module description is instrumented with detailed logging statements. The Logging can be selectively enabled/disabled for specific modules, supporting focussed debugging. The model has been validated thoroughly using unit tests for individual components and shared-memory benchmarks for the whole model. For all simulations results discussed in the following paragraphs, the number of processors in the model(m) and the number of cores per-processor(n) are set to 2 and 4 respectively, to model an 8-core system.

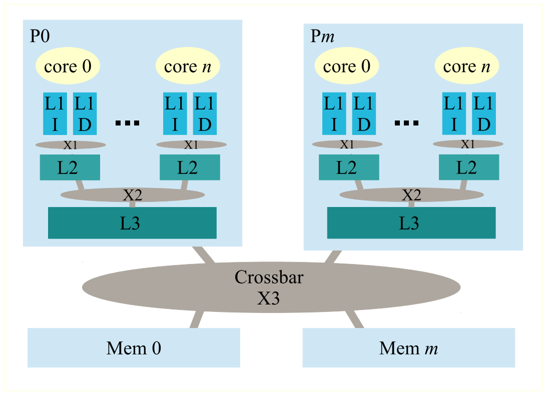

Figure 2: Multi-core system modeled using Sitar.

Table 1: NAS benchmark kernels used as workload and their problem sizes.

Kernel	Problem Size
Embarrassingly Parallel (EP)	2^{16}
Multigrid (MG)	16^3
3-D FFT PDE solver (FT)	32^3
Integer Sort (IS)	$2^{14}+2^{12}$

Workload: Four kernels from the NAS parallel benchmark suite (NPB)(Bailey et al., 1991) are used as workload running on the 8-core model. The kernels and their problem sizes are listed in Table 1. Each kernel is 1 to 2 million instructions long.

Host Configuration: Simulations were performed on a 3.3 GHz Intel machine with 4 physical cores, running Linux (Ubuntu 18.04). The real time for simulating the execution of the complete benchmark kernel on the 8-core model was measured for parallel simulations using 1,2 and 4 OpenMP threads to measure the speedup.

Speedup and Performance: Table 2 summarizes the performance and the speedup obtained for each of the benchmark workloads. The results indicate that a decent speedup is possible with minimal additional modeler effort for parallelizing the simulations. A more detailed scalability evaluation on a many-core system for benchmark designs is necessary and planned as future work.

Table 2: Simulation time and Speedup obtained for simulating the execution of NAS benchmark kernels on the multi-core cycle-accurate simulation model.

Workload (Sim Cycles)	Simulation time in seconds and (Speedup)		
	1 thread	2 threads	4 threads
EP (9781300)	954 (1x)	622 (1.5x)	442 (2.2x)
FT(16578568)	1710 (1x)	1112 (1.5x)	855 (2x)
IS(1434771)	156 (1x)	102 (1.5x)	78 (2x)
MG(13728356)	1452 (1x)	944 (1.5x)	726 (2x)

5 SUMMARY AND CONCLUSIONS

This paper presented Sitar- an open-source cycle-based modeling framework. The key design aspects of Sitar are summarized here:

1. **Cycle-base, Two-phase Execution:** The two-phase, cycle-based execution algorithm used by Sitar makes the simulation deterministic and easy to parallelize. Sitar differs from other cycle-based tools such as Cascade as it uses a two-phase simulation algorithm. The two-phase approach eliminates the need for building a static dependency graph of the components to determine the order of execution among modules. However, there are two execution iterations required in every clock cycle, which may lead to some performance overheads.
2. **Lightweight Simulation Kernel** The two-phase approach makes the simulation kernel simple and extremely lightweight in terms of code size. The kernel comprises about 1000 lines of C++ code, inclusive of comments. Consequently the translated model code which inherits from the simulation kernel classes is also small. The kernel has no external dependencies. The translated code can be compiled to run on any system that supports the gcc compiler and OpenMP. The only dependency that Sitar has is in the translation phase as the translator uses the Antlr V3 library. The Antlr V3 runtime library is bundled along with the Sitar distribution and gets installed automatically by Sitar's installation scripts. The Sitar translation and compilation scripts are written for being run from a Linux terminal, and are well-tested on Ubuntu 18.04 and Ubuntu 20.04.
3. **Domain Specific Modeling Language** Frameworks such as SystemC or Cascade are C++ libraries. The modeler invokes the modeling constructs via C++ macros defined by the framework. This may sometimes lead to verbose model descriptions. Sitar consists of a modeling language that provides several features for easing modeling effort and may produce more concise and readable model descriptions. The descriptions get translated to readable C++ code. The translated code contains line numbers of the corresponding sitar source code to help in debugging compilation-time errors in the embedded C++ code. On the downside, a new user would need to learn the modeling language instead of being able to describe the model solely using a popular programming language like C++ or Python.
4. **Fine-grained Logging Support:** The built-in logging support was found to be one of the most useful features for creating and validation of the Multi-core model. The modeler can instrument each module with detailed log statements. During simulation, it is often desired to view logs from only certain components or during a certain time period in a very long simulation. Saving all logs to a file for later filtering is difficult, as the full log files can quickly reach an enormous size. Thus the fine grained logging control provided in Sitar becomes highly useful for large, complex models. Logs from different modules can be enabled/disabled as and when needed. The log streams from each module can be sent to a single file/screen or can also be redirected to separate files if necessary. Further, all logging can be disabled at compile time if required for a fast simulation in design exploration experiments.

In this paper, we briefly discussed the performance and scalability observed for a particular use case of multi-core modeling. However, a detailed performance review of Sitar and performance comparison with other frameworks such as SystemC is necessary and is planned as future work.

REFERENCES

Akram, A. and Sawalha, L. (2019). A Survey of Computer Architecture Simulation Techniques and Tools. *IEEE Access*, 7:78120–78145.

Bailey, D. H., Barszcz, E., Barton, J. T., Browning, D. S., Carter, R. L., Dagum, L., Fatoohi, R. A., Frederickson, P. O., Lasinski, T. A., Schreiber, R. S., Simon, H. D., Venkatakrishnan, V., and Weeratunga, S. K. (1991). The NAS Parallel Benchmarks Summary and Preliminary Results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, New York, NY, USA. ACM.

Buchmann, R. and Greiner, A. (2007). A fully static scheduling approach for fast cycle accurate systemC simulation of MPSoCs. In *2007 Internatonal Conference on Microelectronics*, pages 101–104.

Dömer, R., Chen, W., and Han, X. (2012). Parallel discrete event simulation of transaction level models. In *17th Asia and South Pacific Design Automation Conference*, pages 227–231.

Fujimoto, R. M. (1990). Parallel Discrete Event Simulation. *Communications of the ACM*, 33(10):30–53.

Fummi, F., Quaglia, D., and Stefanni, F. (2008). A SystemC-based framework for modeling and simulation of networked embedded systems. In *2008 Forum on Specification, Verification and Design Languages*, pages 49–54.

Grossman, J., Towles, B., Bank, J. A., and Shaw, D. E. (2013). The role of Cascade, a cycle-based simulation infrastructure, in designing the Anton special-purpose supercomputers. In *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–9.

Karanjkar, N. and Desai, M. (2022). Sitar: Online repository and documentation (https://nehakaranjkar.github.io/sitar/).

OSCI (2021). SystemC - The language for System-level design, modeling and verification (https://systemc.org/).

Parr, T. (2009). Antlr V3 -ANother Tool for Language Recognition (http://www.antlr3.org).

Ren, P., Lis, M., Cho, M. H., Shim, K. S., Fletcher, C. W., Khan, O., Zheng, N., and Devadas, S. (2012). Hornet: A cycle-level multicore simulator. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 31(6):890–903.

SimPY, T. (2021). SimPY - Discrete-Event Simulation for Python (https://simpy.readthedocs.io).