DecompositionJ: Parallel and Deterministic Simulation of Concurrent Java Executions in Cyber-Physical Systems

CHONG SHUM\textsuperscript{1}, (Student Member, IEEE), WING-HONG LAU\textsuperscript{1}, (Senior Member, IEEE), TIAN MAO\textsuperscript{2}, (Student Member, IEEE), HENRY SHU-HUNG CHUNG\textsuperscript{1}, (Fellow, IEEE), NORMAN CHUNG-FAI TSE\textsuperscript{1}, (Member, IEEE), KIM-FUNG TSANG\textsuperscript{1}, (Senior Member, IEEE), AND LOI LEI LAI \textsuperscript{3}, (Fellow, IEEE)

\textsuperscript{1}Center for Smart Energy Conversion and Utilization, City University of Hong Kong, Hong Kong
\textsuperscript{2}Electric Power Research Institute, China Southern Power Grid, Guangzhou 510700, China
\textsuperscript{3}School of Automation, Guangdong University of Technology, Guangzhou 510006, China

Corresponding author: Chong Shum (chongshum2@gmail.com)

This work was supported by the Hong Kong Research Grant Council under Grant CityU-116013.

ABSTRACT Simulation and performance evaluation of concurrent Java program execution have been difficult due to the lack of proper model and tools. Previous modeling and simulation approaches cannot simultaneously achieve three tasks: (1) support the direct simulation of unmodified multi-thread Java programs; (2) guarantee deterministic simulation results; and (3) offer low-overhead and scalable simulation. This paper first presents a novel simulation model based on the Java memory model. The model axiomatically defines action ordering, relationships, and constraints to ensure the well-formedness and determinism of a simulated execution. Then, based on the model, we implement the DecompositionJ simulation framework (deterministic, concurrent multi-processing simulation for Java programs) which enables the direct-execution simulation of target program by using compiler-based source-to-source transformation and a purposely designed runtime library. The framework is compatible with any JVM that complies with the Java specifications, and does not require manual modifications on the target program code. The performance of the framework has been evaluated with the Grande Java concurrency benchmark suite, results have shown a geometric mean of 98.9\% overhead over all cases, which significantly outperforms full-system simulation techniques.

INDEX TERMS Direct execution simulation, modeling and simulation of concurrent execution, memory models, cyber-physical systems.

I. INTRODUCTION

The use of parallel and distributed computing is pervasive for a wide range of cyberphysical applications. The Java platform, with a virtual machine based approach and efficient just-in-time compilation, remains to be a highly popular mainstream platform for these applications. In this paper, we present a novel model and the DecompositionJ simulation framework (DEterministic, COncurrent Multi-ProCessing SImulation for Java programs) to enable the study and evaluation of multi-thread Java programs. Compared with real-world or testbed experiments, simulation allows the study of program behaviors with unlimited scale and configurability, and also generates reproducible results that are otherwise difficult to obtain in real-world experiments.

The simulation of a cyberphysical system includes physical plants, end-nodes (i.e. the computers that execute distributed programs), and the intermediate networks. Whereas the simulation of application specific plants is beyond the scope of this paper, and mature techniques for network simulation already exist, therefore in this paper, we only focus on the modeling and simulation of the end-nodes. The proposed model comes with the notion of external events to allow the co-simulation of interactions between end-nodes and plants/network.

Many models have been proposed in the literature for simulating the behavior of end-nodes and it is observed that many of them fall into one of the two extremes: either too macroscopic or too microscopic. On the macroscopic side, Grid and Peer-to-Peer Simulators [1]–[5] solely focus on
scalability and performance. They often rely on simplified and highly abstract computation models, based on finite state machines, task graphs, or some specific programmatic patterns. With these highly abstract models, functional and timing realism of each simulated node are sacrificed. In addition, development of abstract models to faithfully represent a target application requires delicate and labor-heavy work. On the microscopic side, architectural simulators [6]–[12] model end-node computation with very high functional and timing realism. Hardware specific details, such as instruction set architecture, processor pipelines, memory system, and cache-coherence protocol are incorporated into the model. Such detailed simulation is prohibitively time consuming, typically suffers from 100–1000× slowdown (compared to native execution) even with the simplified hardware models, and up to 10000× slowdown with the detailed models. Such an enormous performance penalty prevents their usage on simulating cyberphysical systems of practical scale. However, an attractive benefit offered by architectural simulation is that the target program can be directly executed on the simulator without the need of labor-intensive development of computation models.

In this paper, we propose to use an intermediate approach between these two extremes. On one hand, it can achieve relatively scalable and low-overhead simulation; on the other hand, it supports high fidelity modeling of functional and timing characteristics, yet eradicates the need of program modification. This is achieved by using the direct execution simulation techniques: the target program’s code are executed during simulation to emulate its functional behavior. Additional simulation control codes are inserted into the target program to (i) redirect program’s interactions with real systems (i.e. file/network I/O and system clock/timer) to their simulated counterparts, (ii) track the timing of the program, and (iii) control the pace of the target program to achieve time synchronization with other logical processes in the simulation.

Direct execution technique has been used by previous works in High Performance Computing (HPC) and network research communities. For example, SMPI [13] proposed a virtualized implementation of the MPI standard on top of the SimGrid platform, which allows sequential simulation of single-threaded C/C++/Fortran MPI programs using direct execution techniques. Reference [14] simulates ad-hoc wireless routing protocols by (i) redirecting network related system calls to their emulated counterparts, and (ii) privatizing the process memory spaces. Shadow [15] performs sequential direct execution simulation on single-threaded and non-blocking Tor network applications. Later work Shadow-Bitcoin [16] extends Shadow to support multi-threaded C++ application out-of-the-box. Lastly, the Direct Code Execution framework of the well-known NS-3 network simulator [17] takes the traditional library operating system approach, which provides emulated kernel, POSIX and network APIs for supporting simulation of arbitrary C++ applications.

In comparison, DecompositionJ framework contributes to the literature by offering the following features that were otherwise missing:

**Extending direct execution simulation to Java Programs.** Previous works support multi-threaded Fortran/C/C++ programs by means of replacing the threading APIs at system level. However, the same technique does not apply to Java because Java programs do not rely on system level APIs and multi-threading is part of its language specification. In view of this, we propose a simulation model on a language level, which is inspired by the Java Memory Model (JMM) [18]–[20]. The proposed model is generally applicable to arbitrary Java programs. However, in order to guarantee deterministic outcome, the target program is required to be data-race-free. In addition, if the target program is not “purely” Java, i.e., it executes external codes via the Java Native Interface (JNI), then it is up to the users to ensure the external code does not introduce non-deterministic behaviors and deadlocks.

**Modeling of local parallelism.** Previous works either assume that the simulated program is executed on a virtual environment with single processor or each thread is executed by a dedicated processor (NS-3 [17]). In DecompositionJ, the logical processors available in an end-node can be specified and the simulation accounts for thread scheduling, context switching delay, and the consumption of processor time by each logical thread.

**Parallel simulation:** The simulator itself exploits parallelism offered by the host computer. The hardware setting of the host does not constraint the simulated end-node or affect simulation outcomes.

The major challenges in the development of DecompositionJ is to simulate a concurrent Java execution on a virtual multi-processor system, while exploiting the parallelism offered by the underlying host computer as well as ensuring deterministic simulation results. To achieve this, one must rely on a clearly formulated simulation model for concurrent Java execution instead of vague empirical understandings. To our best knowledge, this is the first work that presents such a model. In Section II, we first analyze the behavior of a multi-thread Java program and identify the sources of non-determinism that may arise in a native execution. In Section III, we present an overview of the proposed simulation model that removes these sources of non-determinism. Further details and definitions are then explored in Section IV. Based on the proposed model, Section V presents the techniques for tracking and controlling a direct execution simulation. Implementation details and related tools are discussed in Section VI. Performance evaluation results are shown in Section VII and final conclusion is given in Section VIII.

**II. MULTI-THREAD PROGRAM EXECUTION**

In a single-thread Java execution, the program performs a sequence of actions. The order of actions issued by the single thread is given by the program order → po, a total order over all
actions issued by the same thread. Given a particular control flow path, program order is uniquely defined according to the thread local semantics. A valid/legal execution must exhibit behavior that is consistent with the program order, therefore, the execution of a single thread program is deterministic and repeatable as long as it does not contain unspecified actions such as reading object references and reading external inputs.

In a multi-threaded program, the order of actions issued by the same thread is still governed by the program order, whereas actions issued by different threads are not necessarily in order. Intra-thread actions (e.g. reading/writing local variables) do not require inter-thread ordering since they are independent of actions issued by other threads. Inter-thread actions are further distinguished between normal memory operations (normal read/write of shared variables) and synchronization actions (e.g. volatile read/writes of shared variables, lock/unlock). In Java memory model, only synchronization actions have sequentially consistent semantics, therefore, a total order over all the synchronization actions exists in a Java execution, namely, the synchronization order \( \rightarrow \). Unlike program order, synchronization order is not defined in a way that can be uniquely determined, instead, any ordering that satisfies a set of validity constraints can be used and their corresponding observable behavior are allowed to be displayed in a valid execution. While this relaxation is necessary for compilers and execution environments to optimize runtime performance, it is also the main source of non-determinism in multi-threaded executions.

The following example illustrates the concepts of \( \rightarrow_{po} \), \( \rightarrow_{so} \), and the non-determinism that may arise. For a program consists of three threads as shown in Fig. 1, the main thread is executed first to initialize shared volatile variables \( x, y \), and then Thread-1 and 2 are spawned to accesses \( x, y \). Fig. 2 shows an execution graph of the program, which corresponds to the control flow in which the if-condition at line-6 is evaluated true. Each node on the graph represents an action labeled with its own type, e.g. volatile read/write, and thread spawn. The directed edges represent orders. Program order \( \rightarrow_{po} \) form chains over actions issued by the same thread. Synchronization order \( \rightarrow_{so} \) relates synchronization actions. Since \( \rightarrow_{so} \) is constrained by some validity rules, some \( \rightarrow_{so} \) relationships are fixed and represented by the solid lines. For example, \( \rightarrow_{so} \) must be consistent with \( \rightarrow_{po} \), i.e., if two synchronization actions are issued by the same thread, \( \rightarrow_{so} \) must agree with \( \rightarrow_{po} \). In addition, the first action of a thread must be ordered after the thread’s Spawn action. Other than these constraints, the synchronization order is variable and represented by the dash lines.

The ambiguity in \( \rightarrow_{so} \) allows high-level races between dependent actions: access to \( x \) at action-5/8, and access to \( y \) at action-7/9. As a result, the memory contents, control flow path of each thread, and the timing performance of the program can vary across multiple runs. Fig. 3 shows two valid executions with different orders on the racy actions. In case (a), action-5 was executed before action-8, the read of \( x \) returned value 0 (written by stmt-1) and cause the if statement to branch away from stmt-7. This execution involves a total of 7 actions. In case (b), action-8 was executed first, the read of \( x \) returned value 1 and cause the if statement to fall-through to action-7. The execution involves a total of 8 actions.

A. SOURCES OF NON-DETERMINISM

The causes of non-deterministic/un-repeatable behaviors in multi-thread programs and the corresponding countermeasures are summarized below:

1) **Source**: Actions’ execution order is not deterministically defined and enforced as explained above.

---

**FIGURE 1.** Variable declarations are omitted for clarity. \( x/y \) are volatile variables, \( r1/r2 \) and \( thr1/thr2 \) are local variables.

**FIGURE 2.** An execution graph for the program listed in Fig. 1 (assuming if-condition is evaluated true). \( Wr/Rd \) represent volatile write/read actions. \( Sp \) represents thread spawn. Since intra-thread actions (e.g. access to local variables) cannot detect or affect actions of other threads, they are omitted. Note that \( po \) and \( so \) are transitive, redundant edges are not shown.

**FIGURE 3.** Two valid executions for the program in Fig. 1. (a) final results: \( x=1, y=0, r1=0, r2=0 \); Inter-thread actions: 7. (b) final results: \( x=1, y=1, r1=1, r2=1 \); Inter-thread actions: 8.
Countermeasure: In our model, a total order over all actions is defined deterministically, namely the clock order $\rightarrow_{\mathcal{C}}$. The outcome of the simulation will be repeatable if the clock order is enforced during simulation. However, note that the direct enforcement of a total order prevents the utilization of host parallelism. Therefore, the simulation enforces a partial order derived from the clock order, namely the clock-synchronization order. Simulation result obtained by the enforcement of clock-synchronization order will be the same as if the clock order was enforced, under the condition that the program is data-race-free. In order to enforce clock-synchronization order during simulation, logical time barrier codes are inserted ahead of actions, which postpones an action’s execution until all preceding actions have been completed.

2) **Source:** Thread scheduling is not defined nor controlled by the program, which contributes to the non-determinism of action’s execution order.

**Countermeasure:** Thread scheduling is modeled by introducing logical processors, logical threads, and processor actions to our model. Logical threads participate in the scheduling by executing processor actions (i.e. processor-contend, acquire and release actions). The outcomes of these actions are deterministically defined with a set of calculated relationships. In order to enforce the scheduling behaviors, a logical processor barrier is inserted after each processor release action, which postpones further action execution until the thread has re-acquired a logical processor.

3) **Source:** The outcomes of Java synchronization operations (e.g., lock/unlock, wait/notify) may be non-repeatable even if their order of execution are exactly repeated. For example, when multiple threads contend for a lock, the order of lock acquisition is not defined in JMM.

**Countermeasure:** Java synchronization operations are modeled using lock, wait set, notify, and interrupt actions. The outcomes of these actions are deterministically defined with a set of calculated relationships.

4) **Source:** Interactions between a program and external systems may not be exactly repeatable.

**Countermeasure:** External systems should be replaced with their simulated counterparts, i.e., co-simulators. Interactions with other co-simulators are modeled as external actions, which are covered by the clock order. Similar to other actions, logical time barrier is inserted before the processing of an external message to enforce clock order.

Note that there are other sources of non-determinism unrelated to multi-threading, which are not discussed here. They are caused by operations pertaining to memory management, such as the value of object references/hashcode. These forms of non-determinism are not handled in our model, however they can be easily avoided by using alternative implementations.

III. SIMULATION MODEL OVERVIEW

This section presents an overview of the simulation model. Key components of the model are described, including the relationships between actions and the constraints to ensure a well-formed simulation. From a top level point of view, an simulated execution $E$ is described as a tuple, comprising:

$$(A, T, LP, EP, L, WS, \overset{po}{\rightarrow}, \tau, C)$$

- $A$ - a set of actions.
- $T$ - a set of logical threads.
- $LP$ - a set of logical processors.
- $L$ - a set of locks.
- $WS$ - a set of wait sets.
- $\overset{po}{\rightarrow}$ - program order: a partial order on $A$. The restriction of program order to actions performed by the same thread or same external operation is total.

- $\tau$ - temporal model: $cyc = \tau(y)$ returns the number of processor cycles consumed by action $y$.
- $C$ - clock function: $t = C(y)$ returns the logical time $t$ of an action $y$.

Some elements above are known prior to the simulation, i.e. $LP, EP, \tau$. Logical processors $LP$ and their characteristics, such as context switching delay and clock rate are specified by the user before simulation. External processes $EP$ (i.e. other co-simulators) that may interact with the simulated program are pre-specified. The rank of logical processor and external process are uniquely determined by their ID number for establishing order between external and internal actions. The computation delay function $\tau$ can be obtained prior to simulation via profiling or static analysis. In this thesis, $\tau$ is a simple mapping from an action’s type to a natural number. More sophisticated delay models could be used, however, accurate and efficient modeling of timing characteristics of a virtual machine based execution is in itself a very complicated matter that deserves dedicated research.

The remaining elements are incrementally built during simulation, i.e. $A, \overset{po}{\rightarrow}, T, L, WS, C$. Actions $A$ and program order $\overset{po}{\rightarrow}$ are observed as the simulation proceeds. As mentioned, the target program is first analyzed and then instrumented before simulation. Actions are first identified from the program source code during analysis, followed by injecting the simulation control codes to track for controlling the execution of each action. During direct execution simulation, each thread goes through a particular control path. New actions along these paths are being instantiated and added to $A$ and their order of instantiation gives $\overset{po}{\rightarrow}$. New members of $T, L, WS$ are added as new threads/locks/wait and sets are created during simulation. Lastly, the clock function $C$ computes each action’s time value when they are being instantiated. Details of these elements are introduced in the later section.

---

1 For a set $S$, a binary relation $R$ is the restriction of $R$ to $S$ is defined as: $R|_S = \{(x, y) | x, y \in S \land (x, y) \in R\}$. 

---
A. ACTION TYPES AND ORIGINS
Each action in A is characterized by its type and origin. The action types presented below are inspired by JMM [18] and other related work [21] with modifications to account for wait, notify, and scheduling operation.

- Intra-thread actions (Intra), which is used to model any local operations that are independent of actions in other threads/external events.
- Normal Read/Write (Rd/Wr) and Volatile Read/Write (RDt,WRt) of shared variables.
- Scheduling related actions: Processor Contend/Release/Acquire (PC/PR/PA) and Processor-State (PS).
- Lock related actions: Lock Contend/Release/Acquire (LC/LR/LA) and Lock-State (LS).
- Wait and signaling actions: Wait (W), Notify (N), Interrupt (I), Wait-Set-State (WS), Thread-Wait-State (TW).

Note that the model only covers basic synchronization operations, i.e. locks, wait-set, notify, and interrupt, which provide the basis for constructing advanced synchronization mechanisms, such as semaphore, barriers, conditional variables and read-write locks. However, in order to achieve higher performance, modern implementations of advanced mechanisms typically utilize additional low level synchronization operations such as compare-and-swap and thread-parking/unparking. Since these additional operations are outside the scope of JMM, they are not currently included in our model, and are left for future development.

The origin of an action determines the context to which the action is executed. An action can either be (i) a thread action (thrd) performed by a logical thread of the target program, (ii) an initialization action (init) executed at the beginning of simulation to setup the logical environment, or (iii) an external action (extn) executed as a result of receiving events from co-simulators or external systems.

Each action is defined as a tuple and its elements depends on action’s type and origin. To illustrate the idea, the definitions of the most basic type Intra are observed. When an Intra action is originated from a thread, it is defined as Intra : ⟨u, k, o, th, p⟩. The first three elements are common to all actions regardless of their types and origin: an unique action identifier u, an action type k (which is Intra in this case), the origin of the action o (which is thrd in this case). The remaining two elements are common to thread actions: the issuing thread th and a logical processor p that executes the action.

When an Intra action is executed during initialization, it is defined as Intra : ⟨u, k, o, th, p⟩. It should be noted that the first three elements remain unchanged. Element th is irrelevant to initialization action since it is not issued by logical threads. Element p is relevant since init actions are executed by logical processors. All initialization actions are executed in the processor with least rank, hence p’s value is fixed as p1.

When an Intra action is of external origin, it is defined as Intra : ⟨u, k, o, t, ep⟩. The elements ep, t are common to all external actions: ep is the external process that issues the action; and t is the logical time to which the action is issued and specified by the external process.

The above examples briefly explain the idea of action types/origins using the simplest action type Intra and other action types may carry additional elements. For example, Read/Write actions contain elements m and v representing the memory location (i.e. the variable) and value being read/written. Details for each types of actions will emerge later when they are being discussed in Section IV.

B. CLOCK, CLOCK-SYNCHRONIZATION, AND SYNCHRONIZED-BEFORE ORDER

- Clock order: A total order over all the actions in A. Clock order is derived from the clock function and program order. y1 \rightarrow y2 indicates that y1 is ordered before y2 logically. The notation \rightarrow is used to represent an immediate predecessor:

\[
y1 \rightarrow y2 \iff y1 \neq y2 \land y1 \rightarrow y2
\]

\[
\neg \exists y' \in A, y' \neq y1 \land y' \neq y2 \land y1 \rightarrow y' \rightarrow y2
\]

- Clock-synchronization order: Note that a direct enforcement of \rightarrow essentially serializes all actions and prevents utilization of host parallelism. To avoid this, simulation control code enforces a relaxed \rightarrow, which is a restriction of \rightarrow to synchronization actions only, i.e., excluding Intra, Rd, Wr actions. The rational for this relaxation is that the actual execution order of actions does not necessarily follow \rightarrow as long as the execution result is consistent with one that follows. In that sense, \rightarrow does not need to be enforced on Intra, Rd, Wr actions because (i) Intra actions have no cross-thread dependency, hence their execution only need to agree with program order which is already ensured by the JVM; (ii) dependent Rd/Wr actions can be ordered transitively by program order and the ordering of synchronization actions in a data-race-free program. The \rightarrow of a simulation is analogous to the \rightarrow of a native execution with the difference that \rightarrow is deterministically generated.

- Synchronized-before order is defined as the transitive closure of the union of program order and clock-synchronization order, i.e. \rightarrow = (\rightarrow, \rightarrow)^+. Since \rightarrow is consistent with \rightarrow and \rightarrow is a partial order. When the simulation is executed on a compliant JVM, any actions y1 \rightarrow y2 indicates that y1 is visible by y2.

C. CALCULATED RELATIONSHIPS

When an actions is instantiated during simulation, its outcome is determined based on its relationships with other inter-thread actions. For instance, the value read by a Rd action depends on which Wr action it sees, and this is determined by the read-from relationship \rightarrow. In our model, calculated relationships are defined as logical predicates in the following
fashion: \( y_1 \xrightarrow{\text{rel}} y_2 \iff P(y_1, y_2, E) \), i.e., two actions \( y_1, y_2 \) has a \( \rightarrow \) relationship if and only if the predicate \( P \) is true. All the relationships in our model are summarized below and formal definitions are discussed in Section IV.

- \( rf \rightarrow m \) - Read-from is a relationship from a \( Wr/Wr_i \) action to a \( Rd/Rd_v \) action. \( w \xrightarrow{rf \rightarrow m} r \) indicates that both actions operate on the same variable/memory location \( m \), and the value read by \( r \) is the one written by \( w \).
- \( \text{cap} \rightarrow \) - Contend-acquire-processor is a relationship from a \( \text{processor-content} \) action to a \( \text{processor-acquire} \) action. \( pc \xrightarrow{\text{cap} \rightarrow} pa \) indicates that the processor contention established by \( pc \) is resolved in the subsequent acquisition \( pa \).
- \( \text{rap} \rightarrow \) - Release-acquire-processor is a relationship from a \( \text{processor-release} \) action to a \( \text{processor-acquire} \) action. \( pr \xrightarrow{\text{rap} \rightarrow} pa \) indicates that the processor released in \( pr \) is acquired in \( pa \). (i.e. \( pr \xrightarrow{\text{rap} \rightarrow} pa \iff pa.p_{acq} = pr.p_{rel} \)).
- \( \text{dis} \rightarrow \) - Dispatch is a relationship from a \( \text{processor-acquire} \) action to a thread action. \( pa \xrightarrow{\text{dis} \rightarrow} y \) indicates that the issuing thread of \( y \) has acquired a processor in \( pa \), and the acquired processor is used to execute \( y \).
- \( \text{call}, \text{rud}, \text{call} \rightarrow \text{run} \rightarrow \) - Contend-acquire-lock and release-acquire-lock relationships are similar to \( \rightarrow \text{cap} \rightarrow \) and \( \rightarrow \text{rap} \rightarrow \), but on lock-contend/acquire/release actions, on a per-lock basis.
- \( \text{w} \rightarrow \text{w} \rightarrow \text{w} \rightarrow \) - Wait-Notify is a relationship from a wait action to a notify action. \( w \xrightarrow{\text{w} \rightarrow \text{w}} n \) indicates that the thread suspended to a wait set in \( w \) is signaled by a notify signal \( n \), hence the thread can leave the wait set and resume execution.
- \( \text{wi}, \text{th} \rightarrow \) - Wait-Interrupt is a relationship from a wait action to a Interrupt action. \( w \xrightarrow{\text{wi}, \text{th} \rightarrow} i \) indicates that the thread suspended to a wait set in \( w \) is signaled by an interrupt signal \( i \), hence the thread can leave the wait set and resume execution.

D. WELL-FORMEDNESS CONSTRAINTS

The behavior of a simulation \( E \) subject to a set of constraints. A correct implementation should only generate well-formed simulations that satisfy all the following constraints:

1) Well-formedness of actions and program order
   a) Actions and program order generated in a simulation is faithful to the semantics of target program.
   b) Program order is total over (i) thread actions issued by the same thread, (ii) external actions sharing the same timestamp, and (iii) any initialization actions.

2) Well-formedness of program order and clock order
   a) Clock order is consistent with program order.
   b) Clock order is consistent with clock function.
   c) Clock order is a total order over all actions.

3) Well-formedness of read/write actions
   a) Reads (\( Rd, Rd_v \)) must read from a Write (\( Wr, Wr_v \)) on the same variable, and the read-from relationship must be consistent with \( \xrightarrow{cs} \) and \( \xrightarrow{no} \).
   b) Volatile Read/Write are issued on volatile variables, and normal read/writes are issued on non-volatile variables.

4) Well-formedness of processor related actions
   a) Processor actions in clock order respects mutual exclusion of ownership, i.e., a logical processor can be acquired by at most one thread at a time.
   b) Processor actions in clock order respects contend-acquire-release cycle, i.e., a thread must (i) first contend for a logical processor before acquiring one, (ii) successfully acquire a processor before releasing it, and (iii) releases previously acquired processor before contending again.
   c) A thread action must be dispatched by a processor-acquire action, and dispatch relationship must be consistent with \( \xrightarrow{co} \).

5) Well-formedness of lock related actions
   a) Lock actions and clock order respect mutual exclusion of lock ownership.
   b) Lock actions and clock order respect contend-acquire-release cycle, similar to 4b, but on a per-lock basis.

6) Well-formedness of Wait related actions
   a) Wait actions and clock order respect wait-signal cycle.
   b) The thread issuing a wait or notify action must be holding the lock associated with the wait set.

7) Well-formedness of external actions
   a) External events originated from the same external process have distinct logical time.

During simulation runtime, the instrumented control code is responsible for checking and asserting the consistency predicates to ensure well-formedness. For example, execution of any thread action will be blocked by barrier code until a logical processor is acquired to execute the next action. In the model, the constraints are often used as invariants for determining calculated relationships and the outcome of actions.

IV. MODEL DETAILS AND DEFINITIONS

To explore the details and definitions of the model, we first present components that concern all actions (e.g. \( \xrightarrow{po} \), \( \xrightarrow{co} \), \( \xrightarrow{cs} \), etc.), then focus on different groups of actions (e.g. processor actions, lock actions, wait and signaling actions) together with their associated relationships and well-formedness predicates. Finally, we discuss the special treatments on external and initialization actions.

A. PROGRAM ORDER

The generation of actions by each thread and their program order will not be explicitly modeled, rather, they can be observed and tracked as the direct execution
simulation proceeds. The model, however, does require actions and program order to be faithful to semantics of the original program (well-formedness 1a). In other words, the instrumented simulation control codes should not (i) alter program’s original control flow, (ii) remove actions/operations from the target program, and (iii) add actions/operations that writes the target program’s state. Note that, however, non-deterministic operations, i.e. lock and wait, in the original program need to be replaced by their simulated versions. The simulated counterparts offer the semantics of the original operation but with added determinism, e.g. FIFO semantics for lock contention. Furthermore, in order to build a global total order (i.e. $\rightarrow$), the model requires $\Rightarrow$ to be total when restricted to actions issued by (i) the same thread, (ii) same external event (external actions belong to the same event if and only if they share the same timestamp), or (iii) on initialization actions. This requirement has been mentioned in well-formedness constraint 1b and is defined as:

$$\forall y_1, y_2 \in A. \quad (y_1.th = y_2.th) \lor (y_1.o = init \land y_2.o = init)$$

$$\lor (y_1.o = extr \land y_2.o = extr \land TS(y_1) = TS(y_2))$$

$$\Rightarrow (y_1 \xrightarrow{po} y_2 \lor y_2 \xrightarrow{po} y_1) \quad (1)$$

When the simulation is executed on a compliant JVM, the observed program order is inherently total over actions issued by the same thread. To ensure the same totality over external events and initialization actions, the implementation should ensure that actions issued by each external event are executed on the same native thread.

**B. CLOCK FUNCTION, TIMESTAMPS, AND CLOCK ORDER**

The clock function $C$ assigns a time value to each action for representing its logical time of occurrence in the simulation. $C$ is defined differently for actions of different origin:

1. For external actions, clock function simply refer to the time element $t$ of the action which is assigned by the originating external process:

   $$\forall y \in A. \quad y.o = extr \Rightarrow C(y) = y.t \quad (2)$$

2. For initialization actions, an initial time $t_0$ is assigned:

   $$\forall y \in A. \quad y.o = init \Rightarrow C(y) = t_0 \quad (3)$$

3. For thread actions, $C$ tracks the consumption of processor time and respect the well-formedness constraints that (i) an action is executed after the acting thread has acquired a processor (well-formedness 4c), and (ii) an action is executed after the preceding action in $\xrightarrow{po}$ has been executed (well-formedness 2a, clock function and $\xrightarrow{co}$ is consistent with $\xrightarrow{po}$).

   $$\forall y, pa, y_{po} \in A. \quad y.o = Thrd \land pa \xrightarrow{ds} y \land y_{po} \xrightarrow{po} y \Rightarrow C(y) = max(C(y_{po}), C(pa)+cs) + \frac{\tau(y)}{freq(y,p)} \quad (4)$$

Here, $y_{po}$ is the immediate predecessor action in $\xrightarrow{po}$; $pa$ is the processor-acquire action that dispatches $y$, which must exist in a well-formed simulation (well-formedness 4c); $cs$ is a strictly positive value representing context switching delay and $freq$ returns the clock rate of a processor. The temporal model $\tau$ returns the amount of processor cycles consumed by $y$. A zero value may be returned, such that $y$ may share a time value with its predecessor $y_{po}$.

Note that logical time is insufficient in providing a total order over actions, because processors and external processes may execute multiple actions concurrently. Therefore, the timestamp of an action is defined as a pair:

$$TS(y) = (C(y), y.p/ep) \quad (5)$$

The ordering of timestamps is determined by first comparing time values and then the rank of processor/external process in the case of a tie:

$$TS(y_1) \geq TS(y_2)$$

$$\iff C(y_1) > C(y_2) \lor (C(y_1) = C(y_2) \land y_1.p \geq y_2.p) \quad (6)$$

It should be noted that timestamp ordering is still partial since multiple actions may share the same timestamp, i.e., they are atomic. Note that atomic actions are related by program order in a well-formed simulation. Therefore, the total order, i.e., $\xrightarrow{co}$ is defined by first comparing the timestamps of actions, then their program order in case of a tie:

$$y_1 \xrightarrow{co} y_2$$

$$\iff TS(y_1) > TS(y_2) \lor (TS(y_1) = TS(y_2) \land y_1.p \xrightarrow{po} y_2) \quad (7)$$

There are two observations to explain why atomic actions are related by program order. First observation: atomic actions must share the same origin with two reasons (i) external processes and logical processors have distinct ranks, hence $extr$ cannot have the same timestamp as $thrd$ and $init$ actions; and (ii) $init$ actions has a time value $t_0$ which is strictly less than the time value of $thrd$ actions due to context switching delay.

Second Observation: Well-formedness constraints guarantee actions with the same origin and timestamp must be related by $\xrightarrow{po}$ with three reasons (i) two $extr$ actions with the same timestamp must belong to same external event (well-formedness 7a: different external events has distinct timestamps), hence be related by $\xrightarrow{co}$ (well-formedness 1b); (ii) two $init$ actions must be related by $\xrightarrow{po}$ (well-formedness 1b); (iii) two $thrd$ actions with the same timestamp implies that they are executed on the same logical processor. Due to the mutual exclusion of processor ownership (constraint 4a), these two actions must belong to different threads, i.e., a processor-release action by the first thread and followed by a processor-acquire action on the second thread. Since context switching delay is strictly positive, actions issued by second thread must have a greater timestamp, which contradicts the premise of atomicity. Therefore atomic $thrd$ action must be
issued by the same thread and thus be related by $\rightarrow_{po}$ (well-formedness 1b).

C. READ/WRITE ACTIONS AND READ-FROM RELATIONSHIP

The definition of read/write actions is given as follows (third origin):

- $Wr$, $Wrv$, $Rd$, $Rdv$ : $\{u, k, o, th, p, m, v\}$

Here, $m, v$ represent the memory location and value being written or read. For read actions, $v$ depends on which write action it sees, which is described by the read-from relationship $\rightarrow_{rf}$.

$$\forall r, w \in A. \quad w \rightarrow_{rf} r \implies r. v = w. v$$  (8)

In a well-formed simulation, each read action must see a write on the same location (well-formedness 3a):

$$\forall r \in A. \quad r.k = Rd \lor r.k = Rdv \implies \exists w \in A, w \rightarrow_{rf} r$$  (9)

And the read-from relationship must be consistent with both program order and clock-synchronization order, i.e., the synchronized-before order.

$$w \rightarrow_{rf} r \implies ( w.k = Wrv \land r.k = Rdv \lor w.k = Wrv \land r.k = Rdv \land r.m = w.m \land \neg \rightarrow_{sb} w \land \neg (\exists w'.(w'.k = Wrv \land w'.k = Wrv \land w'.m = r.m \land w'.f \neq w \land \rightarrow_{sb} w')) \text{ (10)}$$

Voluntary reads can only see the write on the same variable that immediately precedes it $\rightarrow_{cs}$. For normal reads, they can see the immediate preceding write in $\rightarrow_{cs}$, or any write on the same variable that is not related to it by $\rightarrow_{sb}$ during the presence of data-race, which is not deterministically defined in our model.

D. PROCESSOR ACTION AND RELATIONSHIPS

To simulate the effects of thread scheduling, logical processors are considered as a type of resource being contended by the threads and four types of processor-related actions are defined: Processor-Contend $(PC)$, Processor-Release $(PR)$, Processor-Acquire$(PA)$ and Processor-State$(PS)$. They are defined as follows:

- **PC** : $\{u, k, o, th, p, \text{th}_{pc}\}$

A Processor-Contend action is executed when a thread becomes ready to execute, i.e., becomes Runnable. Element $\text{th}_{pc}$, provided by thread-local context, represents the contending thread.

To abide to a well-formed contend/acquire/release cycle (constraint 4b), when a $PC$ is executed, the contending thread (i) must not be holding a processor, and (ii) not already contending. Formally:

$$\forall pc, \quad pc.k = PC \implies (\neg \exists pa \in A. pa.\text{th}_{pa} = pc.\text{th}_{pc})$$

$$\land UnreleasedAcquisition(pa, pc)$$

$$\land (\neg \exists pc' \in A. pc'.\text{th}_{pc} = pc.\text{th}_{pc})$$

$$\land UnsettledContention(pc', pc)$$  (11)

Here, $UnreleasedAcquisition(pa, pc)$ determines whether a processor acquired in $pa$ is still being held (yet to be released) at the time $pc$ is executed; and $UnsettledContention(pc', pc)$ determines whether a previous contention $pc'$ is still in effect (yet to be settled):

$$UnreleasedAcquisition(pa, y) \iff pa.k = PA \land pa \neq y \land pa \rightarrow_{co} y$$

$$\land (\neg \exists pr, pr.k = PR \land pr \neq y \land pa \rightarrow_{co} pr \rightarrow_{co} y \land pr.p_r = pa.p_a)$$  (12)

$$UnsettledContention(pc, y) \iff pc.k = PC \land pc \neq y \land pc \rightarrow_{co} y$$

$$\land (\neg \exists pa, pa.k = PA \land pa \neq y \land pc \rightarrow_{co} pa \rightarrow_{co} y \land pa.\text{th}_{pa} = pc.\text{th}_{pc})$$  (13)

- **PR** : $\{u, k, o, th, p, \text{th}_{pr}, pr\}$

A Processor-Release action is executed when a thread is no longer running. Element $\text{th}_{pr}$ represents the releasing thread and $pr$ is the released processor, and they are provided by thread-local context.

Under well-formedness constraints 4b, $PR$ is executed only if: (i) the releasing thread is the acting thread and the holder of released processor; or (ii) $pr$ is an initial processor release at the beginning of the simulation:

$$\forall pr \in A. \quad pr.k = PR$$

$$\implies InitialRelease\_R\_pr$$

$$\lor (pr.\text{th}_{pr} = pr.\text{th} \land pr.p_r = pr.p \land (\exists pa \in A. pa.\text{th}_{pa} = pc.\text{th}_{pc}))$$  (14)

Here, $InitialRelease\_R\_pr$ determines whether $pr$ is the first Processor-Release action during initialization:

$$InitialRelease\_R\_pr \iff pr.o = \text{init}$$

$$\land (\neg (\exists pr'.k = PR \land pr'.p_r = pr.p_r \land pr'.f \neq pr \land pr'.\text{th}_{pr} \rightarrow_{co} pr))$$  (15)

- **PA** : $\{u, k, o, th, p, \text{th}_{pa}, pa\}$

A Processor-Acquire action assigns processor to a contending thread when processor is available. Element $\text{th}_{pa}$ is the acquiring thread and $pa$ is the acquired processor, and they are determined by contend-acquire relationship $\rightarrow_{cap}$ and release-acquire relationship $\rightarrow_{rap}$ respectively:

$$\forall pa, pc \in A. \quad pc \rightarrow_{cap} pa \implies pa.\text{th}_{pa} = pc.\text{th}_{pc}$$  (16)

$$\forall pa, pr \in A. \quad pr \rightarrow_{rap} pa \implies pa.p_a = pr.p_r$$  (17)

Under well-formedness constraints 4b, an acquire action $pa$ is to be executed only if: (i) unsettled processor contention
exists, and (ii) available processor for settling the contention also exists:

\[ \forall p_a, \ p_a.k = PA \implies (\exists p_c \in A. UnsettledContention_p(p_c, p_a)) \]
\[ \land (\exists p_r \in A. UnclaimedRelease_p(p_r, p_a)) \]  
\[ (18) \]

Here, UnclaimedRelease_p(p_r, p_a) determines whether the processor released by p_r is still available (yet to be acquired).

UnclaimedRelease_p(p_r, y)
\[ \iff p_r.k = PR \land p_r \neq y \land p_r \co y \]
\[ \land \neg (\exists p_a. p_a.k = PA \land p_a.p_a = pr.p_r \land p_r \co p_a \co y) \]
\[ (19) \]

As will be discussed, this well-formedness constraint ensures that a PA action is always related to a PC action in \( \cap \), and to a PR action in \( \rightarrow \), such that the elements \( pa.th_{pa} \) and \( pa.p_a \) can always be determined.

- \( PS \) : \{u, k, o, th, p, p_{c\text{unsettle}}, p_{r\text{unclaim}}, p_{a\text{unrel}}\}

A Processor-State action reads scheduler's state, which consists of a set of effective contentions \( p_{c\text{unsettle}} \), available releases \( p_{r\text{unclaim}} \), and effective acquire \( p_{a\text{unrel}} \). These elements are calculated from previous processor actions seen by the PS action:

\[ ps.p_{c\text{unsettle}} = \{ pc | UnsettledContention_p(p_c, ps) \} \]
\[ ps.p_{r\text{unclaim}} = \{ pr | UnclaimedRelease_p(p_r, ps) \} \]
\[ ps.p_{a\text{unrel}} = \{ pa | UnreleasedAcquisition_p(p_a, ps) \} \]
\[ (20) \]

The interactions between processor related actions are determined by \( \cap, \rightarrow, \) and \( \rightarrow \), and are discussed in the following:

- \( \cap \)

Contend-Acquire-Processor relationship encapsulates the logic of a scheduler as it selects the contending action of highest priority to satisfy. Formally,

\[ pc \cap p_a \iff p_a.k = PA \land UnsettledContention_p(p_c, p_a) \]
\[ \land (\neg \exists p_c' \in A. p_c' \neq p_c \land Priority_p(p_c', p_c) \]
\[ \land UnsettledContention_p(p_c', p_a)) \]
\[ (21) \]

Here, Priority_p(p_c', p_c) determines if \( p_c' \) has a higher priority than \( p_c \). To ensure that each PA action is related to exactly one PC action, Priority_p must be defined in such a way that an unique greatest member exists. Here, a simple first-come-

An Acquire-Release relationship is used, i.e. \( Priority_p(p_c', p_c) \iff p_c' \co p_c \). Due to the totality of \( \rightarrow \), an unique greatest member always exists.

- \( \rightarrow \)

Release-Acquire-Processor relationship determines which processor is acquired, among all the available ones. The definition has a similar structure to \( \cap \):

\[ p_r \rightarrow p_a \iff p_r.k = PA \land UnclaimedRelease_p(p_r, p_a) \]
\[ \land (\neg \exists p_r' \in A. p_r' \neq p_r \land Priority_p(p_r', p_r) \]
\[ \land UnclaimedRelease_p(p_r', p_a)) \]
\[ (22) \]

For simplicity, the processor with smaller id is given higher priority, i.e \( Priority_p(p_r', p_r) \iff p_r' \cdot p_r < p_r \cdot p_r \).

- \( \rightarrow \)

In a well-formed simulation, thread action \( y \) is issued only if its acting thread has acquired a processor (constraint 4c):

\[ \forall y \in A, \ y.o = thrd \rightarrow \exists p_a. p_a \rightarrow y \]
\[ (23) \]

Dispatch relationship determines which processor executes a thread action, i.e. element \( p \) of a thread action:

\[ \forall y, p \in A, \ y.o = thrd \land p_a \rightarrow y \iff y.p = p_a.p_a \]
\[ (24) \]

Conceptually, each Processor Acquire action \( pa \) grants the acquiring thread \( pa.th_{pa} \) a processor time slice, which will be ended when the thread executes a Processor Release action \( pr \). Every action executed by \( pa.th_{pa} \) between \( pa \) and \( pr \) are dispatched by \( pa \).

\[ \forall y, p \in A, \ p \rightarrow y \iff p.a.k = PA \land y.o = thrd \land p_a \co y \]
\[ \land p_a.th_{pa} = y.th \land (\neg \exists p_r \in A. p_r.k = PR \land p_r.p_r = y.p \]
\[ \land p_a \co p_r \co y \land TS(p_r) \neq TS(y) \]
\[ (25) \]

The predicate first requires correct type and origin from \( pa \) and \( y \) with \( y.th \) being the acquiring thread in \( pa \). Then the remaining part of the predicate narrows down the relationship to actions that resides within the time slice. Fig. 4 illustrates the idea: consider that thread \( th_1 \) issues a Processor-Acquire action \( pa_1 \) which grants processor \( p_1 \) to thread \( th_2 \), and then \( th_2 \) performs a sequence of actions. In this scenario, the beginning of the time slice is given by \( pa_1 \) in clock order, hence \( th_2 \)'s actions prior to \( pa_1 \) does not belong in this time slice, i.e., \( pr_1 \). Recall that the clock function specifies a positive context switching delay \( cs \), therefore the first action in the time slice has a timestamp strictly greater than \( pa_1 \). The end of the timeslice is determined by the first atomic operation that contains a processor release action.

Processor actions are building blocks of scheduling related operations such as thread spawn, termination, blocking, unblocking, and time slice exhaustion. Fig. 5 shows their composition and control flow. Note that these operations are
designed to be atomic, which prevents any other actions from observing an inconsistent intermediate state of the scheduler. In each operation, a PS action is executed before any PC, PR, PA actions. PS reads the current state of scheduler. The state information is used by consistency predicates to direct the control flow of the operation to enforce the well-formedness of simulation. During a thread spawn operation, the acting thread performs a PA action to grant a processor to the new thread. In a thread termination operation, a PR action is executed. If the release action can resolve a contention, a PA action is executed. When a thread exhausts its time slice, it releases its processor with (PC2) and contends (PC3) for a processor. At the time PA2 is executed, two processor contend actions PC2 and PC3 are visible and unsettled. And as defined in the relationship, the earliest processor contend action PC2 has the highest priority. Finally, the released processor is assigned to th1 when th2 terminates.

E. LOCK ACTION AND RELATIONSHIPS

Lock Actions: Similar to processors, locks can also be contented, released and acquired by threads. Similar well-formedness predicates and calculated relationships are defined for lock actions. The major difference is that processors are homogeneous resources with contentions resolvable by any available processor. Locks are heterogeneous and contentions are specific to a lock instance.

- LC : \( \langle u, k, o, th, p, th1, l \rangle \)

A Lock-Contend action is part of a lock, interrupt or notify operation. Elements th1lc represents the containing thread and l represents the target lock, and both are determined by thread local context. In a well-formed simulation (well-formedness 4b), a LC action is executed only if the containing thread is not already holding or containing the target lock.

\[
\forall lc \in A. \quad lc.k = LC
\]

\[
\implies (\neg \exists la \in A. \quad la.thla = lc.thlc \land la.l = lc.l) \land UnreleasedAcquisition_{1}(la, lc) \land UnsettledContention_{1}(lc', lc)
\]

where UnreleasedAcquisition_{1}(la, y) and UnsettledContention_{1}(lc, y) are defined as:

\[
UnreleasedAcquisition_{1}(la, y) \iff la.k = LA \land la = y \land la \rightsquigarrow y \land (\neg \exists lr \in A. \quad lr.k = LR \land lr \neq y \land lr \rightsquigarrow lr \rightsquigarrow y \land lr.l = la.l)
\]

\[
UnsettledContention_{1}(lc', lc) \iff lc'.k = LC \land lc.l = lc.l \land lc' \neq lc \land lc' \rightsquigarrow lc \rightsquigarrow lc.l = lc.l
\]
UnsettledContention_{L}(lc, y) \iff lc.k = LC \land lc \neq y \land lc \overset{co}{\rightarrow} y \\
\land (\neg\exists la \in A. la.k = LA \land la \neq y \land la \overset{co}{\rightarrow} la \overset{co}{\rightarrow} y \\
\land la.thla = lc.thlc \land la.l = lc.l) \quad (28)

\bullet \ LR : \langle u, k, o, th, p, th_{fr}, l \rangle

A Lock-Release action is part of an unlock and wait operation. Elements th_{fr} is the releasing thread and is determined in thread local context. Well-formedness 4b requires that (i) it is an initial lock release upon lock creation; or (ii) the releasing thread is the acting thread and the holder of the released lock. Formally,

\forall lr \in A. lr.k = LR \implies InitialRelease_{L}(lr) \land
\forall lr \in A. lr.k = LR \implies InitialRelease_{L}(lr)
\land (lr.thfr = lr.th \land \exists la' \in A. la'.thla = lr.thfr \\
\land la'.l = lr.l \land Unreleased Acquisition_{L}(la', lr)) \quad (29)

Here, InitialRelease_{L}(lr) determines whether lr is the first Lock-Release action for the lock lr.l.

InitialRelease_{L}(lr) \iff \neg(\exists lr' \in A. lr'.k = LR \\
\land lr' \neq lr \land lr' \overset{co}{\rightarrow} lr \land lr'.l = lr.l) \quad (30)

\bullet \ LA : \langle u, k, o, th, p, th_{la}, l \rangle

A Lock-Acquire action is executed to grant a lock to a contending thread. Element th_{pa} is the thread that acquires the lock, which is determined by the \overset{cal,l}{\rightarrow} relationship. Element l is the target lock, which is determined by thread local context. In a well-formed simulation, a LA action is executed only if (i) unsettled lock contention exists; and (ii) the target lock is available. Formally,

\forall la \in A. la.k = LA \\
\implies (\exists lc \in A. lc.l = la.l \land UnsettledContention_{L}(lc, la)) \\
\land (\exists lr \in A. lr.l = la.l \land UnclaimedRelease_{L}(lr, la)) \quad (31)

Here, UnclaimedRelease_{L}(pr, pa) determines whether the lock released by lr is still available by the time when pa is executed.

UnclaimedRelease_{L}(lr, y) \iff lr.k = LR \land lr \neq y \land lr \overset{co}{\rightarrow} y \land (\exists la \in A. la.k = LA \\
\land la \neq y \land la \overset{co}{\rightarrow} la \overset{co}{\rightarrow} y \land la.l = lr.l) \quad (32)

In a well-formed simulation, a LA action always relates to a LC action in \overset{cal,l}{\rightarrow} and a LR action in \overset{cal,l}{\rightarrow}. The \overset{cal,l}{\rightarrow} relationship determines the acquiring thread:

\forall la, lc \in A. lc \overset{cal,l}{\rightarrow} la \implies la.thla = lc.thlc \quad (33)

\bullet \ LS : \langle u, k, o, th, p, l, lc_{unsettle}, lr_{unclaim}, la_{unrelease} \rangle

A Lock-State action is a read access to the lock’s states. The states include the set of effective contentions lc_{unsettle}, available releases lr_{unclaim}, and effective acquisitions la_{unrelease}.

\forall ls, lc_{unsettle} \iff lc_{unsettle} = \{lc | lc.l = ls.l \land UnsettledContention_{L}(lc, ls) \}
\land (lr_{unclaim} = \{lr | lr.l = ls.l \land UnclaimedRelease_{L}(lr, ls)) \land ls.la_{unrelease} \\
= \{la | \exists l.l = ls.l \land Unreleased Acquisition_{L}(la, ls)\} \quad (34)
\land \overset{cal,l}{\rightarrow}

The Contend-Acquire-Lock relationship describes the logic of lock arbitration. The simple first-come-first-served principle is used:

lc \overset{cal,l}{\rightarrow} la \iff la.k = LA \land lc.l = la.l \\
\land UnsettledContention_{L}(lc, la) \\
\land (\exists lc' \in A. lc.l = lc'.l \land Priority_{L}(lc', lc) \\
\land UnsettledContention_{L}(lc', la)) \quad (35)
\land Priority_{L}(lc', lc) \iff lc' \overset{co}{\rightarrow} lc \quad (36)
\land \overset{cal,l}{\rightarrow}

The release-acquire-lock relationship share similar meaning as \overset{rap}{\rightarrow}. Locks are heterogeneous resources and there is only one unclaimed release for each lock, whereas for processors, multiple homogeneous unclaimed processors can be selected for acquisition. Since the uniqueness of unclaimed lock release is guaranteed, \overset{cal,l}{\rightarrow} does not contain prioritization predicates. Formally,

lr \overset{cal,l}{\rightarrow} la \iff la.k = LA \land lr.l = la.l \\
\land UnclaimedRelease_{L}(lr, la) \quad (37)

Fig. 7 shows the composition and control flow of the Lock and Unlock operations. It should be noted that these operations contain processor related operations. For example, an unsuccessful attempt to acquire a lock results in blocking of the acting thread; conversely, an Unlock operation could cause a previously blocked thread to unblock.

F. WAIT, NOTIFY, INTERRUPT ACTIONS AND RELATIONSHIPS

The Wait, Notify, and Interrupt operations are thread coordinating mechanisms. A thread performs Wait operation to suspend itself until further notice. During the suspension, the thread is assigned to a wait set which contains all the suspended threads waiting for the same notification signal. The Notify operation signals a targeted wait set to cause one of its member to resume if the wait set is not empty. The interrupt signal targets a specific thread and causes it to leave any wait set it belongs to and resumes execution. Furthermore, wait and notify operations are used in conjunction with an associated lock to protect concurrent access to the wait set, the associated lock must be held when performing wait/notify operations, and be released upon wait suspension.
For a Wait action, element \( s \) represents the targeted wait set and \( th_w \) represents the waiting thread, with all elements provided in thread-local context. Well-formedness constraint 6a requires that (i) a wait action can only be applied to the acting thread itself, and (ii) the thread has not already been waiting, therefore:

\[
\forall w \in A. \quad w.k = W \implies w.o = thrd \land w.th = w.th \land -\exists w' \in A. \quad w'.th = w.th_w \land \text{UnsignaledWait}(w', w) \tag{38}
\]

Here, \( \text{UnsignaledWait}(w', w) \) determines whether a preceding wait action \( w' \) is still in effect (yet to be signaled by Notify or Interrupt actions) by the time \( w \) is executed;

\[
\text{UnsignaledWait}(w, y) \iff w.k = W \land w \neq y \land w \xrightarrow{co} y \\
\land -\exists n \in A. \quad n.k = N \land w \xrightarrow{wn} n \land n \xrightarrow{co} y \\
\land -\exists i \in A. \quad i.k = I \land w \xrightarrow{wt} i \land i \xrightarrow{co} y \tag{39}
\]

- \( N : \langle u, k, o, th, p, th_{tg} \rangle \)

For a Notify action, the target wait set \( s \) is provided by thread-local context.

Well-formedness 6b requires the lock associated with the target wait set must be held when executing notify action.

\[
\forall n, n.k = N \implies \exists la, la.l = \text{assoL}(n.s) \land la.th_la = n.th \\
\land \text{UnreleasedAcquisition}(la, n) \tag{40}
\]

- \( I : \langle u, k, o, th, p, th_{tg} \rangle \)

For an Interrupt action, the target thread \( th_{tg} \) is provided by thread-local context and there is no specific well-formedness constraint on interrupt actions.

- \( WS : \langle u, k, o, th, p, s, w_{\text{unsignaled}} \rangle \)

A Wait-set-State action is a read access to the list of waiting threads \( w_{\text{unsignaled}} \) in wait set \( s \).

\[
w_{\text{unsignaled}} = \{ w \mid w.s = ws.s \land \text{UnsignaledWait}(w, ws) \} \tag{41}
\]

- \( TW : \langle u, k, o, th, p, th_w, w_{\text{unsignaled}} \rangle \)

The Thread-Wait-State action determines which wait set a thread belongs to. This action is used during interrupt operation. The element \( th_w \) represents the target thread and \( w_{\text{unsignaled}} \) represents the effective wait actions on \( th_w \). If a thread does not belong to any wait set, \( w_{\text{unsignaled}} \) will be empty.

\[
w_{\text{unsignaled}} = \{ w \mid w.th_w = tws.th_w \land \text{UnsignaledWait}(w, tws) \} \tag{42}
\]

• \( wi.th \rightarrow w.th \)

The Wait-Notify relationship indicates that a Wait action is resumed by a Notify action and is defined as:

\[
w \xrightarrow{wn} n \iff n.k = N \land w.s = n.s \land \text{UnsignaledWait}(w, n) \\
\land -(\exists w' \in A. \quad w'.s = n.s \land w' \neq w \land \text{Priority}(w', w) \\
\land \text{UnsignaledWait}(w', n)) \tag{43}
\]

Firstly, the relationship requires matching action types and target wait set. Secondly, the wait action has not been resumed. Lastly, when there are multiple threads in the wait set, the wait action of highest priority is selected.

- \( w.th \rightarrow n.i \)

The Wait-Interrupt relationship indicates that a Wait action is resumed by an Interrupt action and is defined as:

\[
w \xrightarrow{wn} w.th \iff i.k = I \land i.th_{tg} = w.th_w \\
\land \text{UnsignaledWait}(w, i) \tag{44}
\]

This relationship requires matching types and targeted thread, and the wait action has not been resumed. It is noted that prioritization is not necessary since there is only one un-signaled wait action for each thread in a well-formed simulation.

Wait, Notify, and Interrupt actions are building blocks of their corresponding operation as shown in Fig. 8.

In a wait operation, the acting thread’s interrupt flag and associated lock’s state are first checked. Wait operation will be aborted with exception if the flag is set or the associated lock has not been acquired. Then, the consistency predicates are checked to ensure the acting thread does not resided in any wait set. After that, Wait action is issued, and finally, Unlock and Block operations are performed to release the associated lock and suspend the acting thread.

For a Notify operation, the associated lock is first checked, followed by a Notify action. If a waiting thread is resumed, i.e., \( w' \xrightarrow{wn} n \) exists, a lock operation will be performed on behalf of the resumed thread. For an Interrupt operation, the acting thread immediately executes an Interrupt action.
If the targeted thread is resumed, i.e., \( w_i \xrightarrow{wi,th} i \) exists, a lock operation will be performed on the associated lock on behalf of the resumed thread. Finally, if the targeted thread is not waiting, its interrupt flag will be set. In addition to the Wait operations, \( \text{Thread.sleep()} \) and \( \text{thread.join()} \) are also implemented with wait and signaling actions and can be considered as special cases of Wait operation.

### G. EXTERNAL ACTIONS

Messages can be exchanged during co-simulation to model the interactions between systems. A message can either be outgoing (sent from the simulated program to other co-simulators) or incoming. Outgoing messages are being interpreted and handled by the receiving co-simulators, the following discussion on external actions only concerns incoming messages.

When an incoming message is received, a handler routine will execute the corresponding external actions. For example, if the message indicates an update of sensor value, then the handler routine will perform a volatile write action on the memory location representing the sensor.

External actions are different from thread actions in the following aspects:

1) External actions do not possess attributes such as processor and thread IDs \( p, th \). Instead, an external process ID \( ep \) is given to represent the source co-simulator. It should also be noted that the dispatched relationship does not apply to external actions.

2) Not all action types can be executed in the context of an external action. Specifically, Processor Release and Lock Release actions \( (PR, LR) \) can only be executed by a thread that holds the processor or lock; Wait and Notify actions \( (W, N) \) can only be executed by a thread which holds an associated lock.

3) The timestamp of an external action is defined as \( \langle ep, t \rangle \), where \( t \) is the time of the incoming message.

Note that multiple actions can be executed by a message handler, i.e. an external operation. An external operation is atomic, i.e., it contains a sequence of external actions ordered by \( \Rightarrow \) and share the same timestamp. In a well-formed simulation, external actions must either have different timestamps or be ordered in program order to ensure that the clock order is total. Formally:

\[
\forall y_1, y_2 \in A. y_1.o = \text{extn} \land y_2.o = \text{extn} \Rightarrow TS(y_1) \neq TS(y_2) \lor (y_1 \xrightarrow{po} y_2 \lor y_2 \xrightarrow{po} y_1)
\]

To satisfy this well-formedness constraint in the implementation, messages originating from the same co-simulator must have distinct time value or provide additional information to order two messages under the same time value.

Other constraints on external actions include (i) the external process ID does not overlap with processor IDs, and (ii) they are executed after initialization (messages with a timestamp less than initial time will be ignored).

\[
\forall y_1, y_2 \in A. y_1.o = \text{extn} \land y_2.o = \text{extn} \Rightarrow y_1.ep > y_2.p \lor y_1.ep < y_2.p
\]

\[
\forall y.y.o = \text{extn} \Rightarrow TS(y) > \langle t_0, p_1 \rangle
\]

### H. INITIALIZATION ACTIONS

The initialization phase is an atomic operation that prepares the simulated environment before being seen by the simulated program. Firstly, an initial \( PR \) action is executed for each logical processor; then a \( PC \) action is executed on behalf of the program’s main thread; finally, it ends with a \( PA \) action to grant a processor to the main thread. In a well-formed simulation, only one processor-release action is executed for each logical processor during initialization.

\[
\forall pr_1, pr_2 \in A. pr_1.k = PR \land pr_2.k = PR \land pr_1.o = \text{init} \land pr_2.o = \text{init} \land pr_1.p_r = pr_2.p_r \Rightarrow pr_1 = pr_2
\]

### V. TRACKING AND CONTROLLING SIMULATION

Based on the proposed model, operational mechanisms are developed to keep track of the simulated execution state and to enforce the ordering of actions during simulation runtime. Here, we adopt some techniques commonly used in the research field of deterministic recording/replay/debugging, which also involves tracking and controlling of program executions. The following subsections first describe the data structure (metadata) used for tracking the execution state during runtime. Then, the techniques to enforce
clock-synchronization order $\rightarrow$ and committing actions are presented. Finally the method for handling external events is described.

A. SIMULATION METADATA

The simulation metadata is a global data structure that records the state of a simulated execution, and is shared and accessible by all the threads during simulation runtime. Metadata provides the necessary information for evaluating consistency predicates, timestamps of actions, and calculated relationships between actions. Its components are listed below:

- **Processor States:** For each processor, metadata maintains its ID, logical clock, the latest dispatched thread, operating frequency, context switching delay, and an idle flag.
- **External Event Queue:** This includes messages received from external processes and sorted according to their timestamps.
- **Logical Scheduler:** This includes a set of active Threads and a processor contention queue. These information allows the evaluation of $\text{UnsettledContention}_P$ (members of the contention queue) and their priority (order of contention queue).
- **Locks:** For each lock, metadata keeps track of its locked/unlocked state, latest acquiring thread, and a contention queue.

These components allow the evaluation of $\text{UnreleasedAcquisition}_L$ (executed by the latest owner of locked locks), $\text{UnclaimedRelease}_L$ (executed by the latest owner of unlocked locks), $\text{UnsettledContention}_P$ (the contention queue), and $\text{priority}_{LC}$ (order of contention queue).

- **Logical Threads:** For each logical thread, metadata maintains its ID, the latest acquired processor and timestamp, the lock it contends, and the wait set it resides in.
- **Wait sets:** For each wait set, metadata contains its wait queue and the associated lock. This allows the evaluation of $\text{UnsignaledWait}_W$.

Since the metadata is a shared data structure, a metadata lock is used to prevent concurrent access leading to inconsistent states. A thread must hold the metadata lock when accessing metadata.

B. TRACKING AND CONTROLLING LOGICAL THREADS

During simulation runtime, each logical thread is backed by a native Java thread. The timestamps of the actions performed by each thread is tracked and the progression of threads are controlled to ensure the well-formedness of the simulation.

1) **PROCESSOR BARRIERS**

In a well-formed simulation, thread actions can be executed only when the acting thread has acquired a logical processor since the timestamp of a thread action depends on the dispatching $PA$ action. Despite this constraint applies to all thread actions, it is not necessary to enforce it on every actions since the predicate remains true until an acting thread releases its acquired processor. Therefore, processor barrier code is instrumented at the beginning of $thread\_run()$ and at the end of every processor release operations, excluding the thread termination operations. When a thread reaches a processor barrier, it will be refrained from further action executions until a logical processor is acquired. As shown in Fig. 9, the acting thread will wait on a per-thread conditional variable if an acquired processor is not observed in the metadata. When a processor is later acquired, the thread that performs the $PA$ action will signal and release the blocked thread from processor barrier. Note that the metadata lock must be acquired before entering barrier, and released while waiting inside or leaving the barrier.

2) **DETERMINE AND ENFORCE CLOCK-SYNCHRONIZATION ORDER**

The timestamps of a thread actions are tracked using the acquired processor’s logical clock in the metadata. The clock of the processor is set to the $C(pa) + cs$ when a thread is being released from processor barriers, as shown on the left side of Fig. 9. After leaving the processor barrier, the processor’s clock value will be updated by the acting thread each time a new action is reached. Essentially, processor clock tracks the timestamp of the next action to be performed by the running thread. Note that, frequent updates of clock values to the metadata is a scalability hazard. Therefore, to reduce performance impact, each logical thread maintains a local copy of the clock and updates the metadata only when (i) it reaches a logical time barrier, or (ii) a period of logical time (10% of the quantum) has passed.

With timestamp information, $\rightarrow$ order can be enforced by inserting logical time barriers ahead of all synchronization...
actions. As shown in Fig. 10, an acting thread first updates the logical clock when it reaches a time barrier. It then waits until all actions with lesser timestamps are executed, which can be determined by comparing the action’s timestamp with the clock of other non-idle processors and also with the timestamps of all external events.

In a co-simulation environment, it is also necessary to ensure that no external events with lesser timestamps will arrive afterwards. For a co-simulation complying with the High-Level-Architecture (HLA, IEEE-1516.2010) standard, a thread can invoke the NextMessageRequest service to obtain the minimum timestamped external event from all other federates, and wait until a TimeAdvanceGrant is issued by the runtime infrastructure. After receiving the grant, the thread will re-enter time barrier since external events with timestamps less than the current action may be received during NextMessageRequest.

C. COMMITTING ACTIONS

After a thread is being released from the logical time barrier, it enters an exclusive action critical section. Actions from other threads will be blocked by logical time barriers for as long as this present thread remains within the critical section. This allows the present thread to execute its action(s) atomically. The critical section ends when (i) the acting thread increases its processor’s clock (e.g. moving on to the next action/operation), or sets the processor’s idle flag, and (ii) the acting thread exits metadata guard. For non-blocking actions/operations, they can be directly executed within the critical section as shown in Fig. 10.

Blocking operations such as Lock and Wait can not be executed within the critical section because blocking prevents the exit of metadata guard at the end of the critical section. Without exiting the guard, no other threads can continue further execution and deadlock will be formed. A trivial solution is to execute the blocking action after exiting the guard, however, this could lead to incorrect ordering of action execution. To illustrate the problem, consider a scenario with two logical threads running on separate logical processors and Thread-1 has a Wait operation ordered before thread-2’s Notify operation. An incorrect synchronization could occur as shown in Fig. 11: Thread-1 first sets processor-1 as idle (assuming no contending threads) and then exits metadata guard. Then, Thread-2 races to execute the Notify operation before Thread-1 finishes its Wait operations. The racing is made possible because (i) Thread-1 has exited the guard, hence Thread-2 can enter metadata guard, and (ii) Thread-1 has marked its processor idle, and Thread-2 now having the minimum clock value can pass through time barrier. The resultant simulation is inconsistent with the $\text{cs}$ order.

The root of the above problem is that the Wait operation is executed outside the correctly synchronized critical section and that results in unsynchronized and freewheeling behaviors. To solve this problem, a dedicated attribute freewheeling-thread is introduced to the metadata. If a logical thread needs to execute actions outside the exclusive critical section, e.g. blocking actions, it must mark itself as the freewheeling-thread before exiting metadata guard. Any other threads that enters metadata guard must then delay their execution until observing the completion of freewheeling-thread’s operation. Reusing the scenario in Fig. 11, Thread-2 will now spin in a loop until the freewheeling-thread has entered blocked state. This spinning wait behavior is appended to both metadata guard enter and metadata guard wait routines as shown in right side of Fig. 11.

D. HANDLING EXTERNAL EVENTS

Depending on the implementation, external messages from other simulators may be pre-processed differently. Our implementation uses an HLA compliant Runtime-Infrastructure (RTI) which handles the low level mechanics for simulator communications and synchronizations. When a message is received from a peer simulator, it is first stored in a buffer managed by the RTI, and later moved to the external event queue in the metadata - this is achieved when executing the rtiamb.tick() method in the logical time barrier. The events on the External Event Queue are sorted according to their timestamps and executed by an External Event Thread. The main body of External Event Thread is a loop. In each iteration, the thread first waits at the logical time barrier until
the next external event has the minimum timestamp, and then executes the actions of that event. The operation principle of an external event thread is similar to a logical thread but with the following differences: (i) external operations does not contain PR/LR/W actions and hence the external event thread will not be blocked as a result of executing external actions; (ii) external events does not require a logical processor for execution and hence it will not execute processor barrier code; and (iii) external event thread will wait at an external event barrier when the event queue is empty, and be released when new event arrives.

VI. IMPLEMENTATION AND TOOLS
Static analysis is performed to identify actions in the target program, then source-to-source transformation is performed to inject simulation control codes to the target source. The output of the transformation is a direct execution simulator of the original program. Since the simulator is itself Java-based, it is compatible with any compliant JDK, JVM and their associated development tools such as debuggers, profilers, and IDEs. Alternative implementation approaches such as bytecode level analysis/instrumentation or modification of JVMs are also possible, and these lower level approaches are likely to provide better runtime performance. However, lower level implementations would suffer from platform lock-in and limited tool support.

A. SIMULATION RUNTIME LIBRARY
The simulation monitoring and control mechanics presented in Section V are implemented in a purposely designed simulation runtime library. Calls to the runtime library are then instrumented to the target program code by the transformer. To simplify the transformation process, commonly used Java synchronization operations (e.g., Object.wait/notify/notifyAll, Thread.start/sleep/yield/join/interrupt/interrupted/isInterrupted/isAlive/getState, monitor enter/exit, System.currentTimeMillis/nanoTime, etc.) are pre-developed and included in the library. Transformation of these operations can be achieved simply by replacing the operations with a call to the runtime library.

B. TARGET SOURCE ANALYSIS AND TRANSFORMATIONS
The JastaddJ/ExtendJ compiler framework [22] [23] is used to perform the analysis and transformation on the target program, with the procedures described in the following:

1) Generate abstract syntax tree (AST) on all the compilation units of the target program.
2) Identify nodes in the AST that indicates synchronization actions and then rewrite those nodes into their instrumented versions. Specifically, (i) for volatile variable access, metadata guard enter and logical time barriers are inserted before the access, and metadata guard exits are inserted at the end; (ii) for Synchronization Blocks and Synchronization Methods, they are replaced with a try-finally block that begins with an monitor enter and ends with a monitor exit operation; and (iii) for operations with instrumented versions, they are replaced with a call to the corresponding operations.

3) After transforming the synchronization actions, the remaining nodes in the AST only contain intra-thread actions. Though synchronization control does not apply to Intra-thread actions, it is still required to track their consumption of processor time. For this purpose, multiple intra-thread actions can be lumped into one to reduce tracking overhead, given that they belong to the same basic block, i.e., a linear sequences of actions with one entry and exit point. Therefore, for each Body Declaration including method, constructor, static initializer and instance initializer bodies, precise intra-procedural control flow analysis is performed using the methods proposed by [24]. Based on the analysis results, the basic blocks are identified and logical clock advancement codes are inserted at the beginning of each block. The transformed AST is then rewritten into Java source which can then be compiled using JDKs and executed on JVMs.

VII. EVALUATION
The modeling fidelity, performance, and scalability of DecompositionJ are evaluated by simulating the execution of Parallel Java Grande Benchmark Suite [25]. The benchmark suite contains three sections of tests with different levels of complexity. Section I of the benchmark suite consists of low-level micro-benchmarks which do not represent realistic applications, hence are not evaluated. Section II consists of kernels that reflect the type of computation which can be found in most computationally intense multi-thread numerical applications. Lastly, Section III contains larger codes for real numerical applications.

A. MODELING OF PROGRAM BEHAVIORS
The simulation of Crypt and SOR in Section II are presented to demonstrate the modeling capability of DecompositionJ. Both benchmarks are simulated under various configurations: 2-LT/2-LP (2 logical threads executed on 2 logical processors); 4-LT/2-LP, and 4-LT/4-LP. In all configurations, the processor frequency is 2GIPS, context switching delay is 100K cycles, and scheduling quantum is 50ms.

1) The Crypt benchmark performs IDEA encryption/decryption on a data array. This is a typical example of a highly parallel task: each thread operates independently on its dedicated array segment, and no inter-thread communication is required except for the indication of completion. Fig. 12 shows thread schedules in Crypt simulation. The task has four phases (e.g. generation of data array, encryption, decryption, and validation). The generation and validation of data array are sequential operations performed by the main thread, while the encryption...
FIGURE 12. Gantt chart for the Crypt benchmark.

and decryption are performed in parallel by multiple threads.
In 2-LT/2-LP or 4-LT/4-LP configurations, a thread issues a PR and a PC action when it exhausts its assigned quantum (50ms). Since there are enough processors for all the threads, the processor contention queue will be empty, hence, the thread will be re-assigned to the same processor and context switching is not required. It is obvious that encryption/decryption time is only half for the 4-LP configuration.

In a 4-LT/2-LP configuration, the four logical threads share the two logical processors with context switching, resulting in an additional context switching delay of 1.45ms.

2) SOR benchmark performs 100 rounds of successive over-relaxation on a randomly generated matrix. In contrast to Crypt, thread operations are dependent and require inter-thread coordinations. The kernel contains an outer loop over 100 rounds and two inner nested loops. During each round, the inner loops updates the elements of the principle array based on previous values and their neighboring elements. To wait for the computation of neighboring elements, a thread stays in a spinlock until it observes the completion flag being set by other threads.

Since the workload on each thread is balanced, the usage of spinlock presents low synchronization overhead when enough processors can be allocated to each thread. However, in the 2-LP/4-LT configuration, if a thread is not being dispatched in a scheduling round, its dependent threads will be blocked for the remaining time of their quantum, resulting in a drastic increase of overhead. The same phenomenon is observed in both native and simulated executions.

The examples above show the ability for DecompositionJ to model target program’s functional and timing characteristics. Compared to Grid and Peer to Peer simulators, DecompositionJ not only allows the study of overall program behavior by observing its I/Os, but also offers realistic insights to low level details for allowing the fine tuning of program performance.

B. PERFORMANCE AND SCALABILITY
All benchmarks in Section II and III are simulated to evaluate the performance and scalability of DecompositionJ. The wall-clock time consumed by native and simulated executions are measured and compared. These tests are then repeated with different numbers of threads (1, 2, 4, and 8).

Both native and simulation programs are built with Oracle JDK 7u55 64-bit, which uses HotSpot 64-bit Server VM 24.55-b03. The test machine is equipped with Intel Xeon E5-2620 processor, 32 GBytes main memory, and run on 64-bit Windows 7. The simulated machine has 8 logical processors. All cases are repeated 10 times to obtain the mean result. The determinism of simulation have also been verified by checking the repeatability of all synchronization actions in all simulated executions and results are shown in Table-1.

In Section II, the simulation of Crypt, Fourier Series benchmarks exhibit low overhead due to the highly parallel nature of their tasks. With little inter-thread communication, synchronization actions in the program are sparse, therefore, heavy-weight simulation control codes such as logical-time barriers are rarely executed.

Sparse Matrix Multiplication (SparseMatmult) is also a highly parallel benchmark, which is reflected by its lower
synchronization/processor action counts. However, it has a long sequential initialization phase for generating the matrix inevitably contains large amount of branching. This results in a large amount of instrumented control codes to track the elapse of processor time.

On the contrary, LU Factorization (LUFact) and SOR benchmarks exhibit higher overheads due to their frequent inter-thread communications. In LUFact, threads achieve synchronization by using a tournament barrier scheme which are implemented with repeated thread yielding. This generates an excess amount of processor actions, which invoke heavy-weight logical time barrier, processor barrier, and scheduling codes. In SOR, the overhead comes from the usage of spinlocks which repeatedly issues volatile access to shared memory locations. Each volatile access causes the simulation thread to contend for metadata lock and enters logical time barrier. In both LUFact and SOR benchmark, the percentage overhead increases with the number of threads. This indicates that the access to shared simulation metadata has become a bottleneck due to large amount of synchronization actions.

Section III contains larger code tests that are intended to represent real numerical applications. ModDyn Benchmark models the dynamic N-body particle interactions, MonteCarlo benchmark simulates financial pricing, and Ray tracer benchmark performs rendering of 3D spheres. Both MolDyn and MonteCarlo benchmarks contain large amount of control flow branches which cause a large amount of instrumented code to track the elapse of logical time. In addition, MonteCarlo simulation contains inefficient uses of volatile variable in generating random numbers and thus results in large amount of synchronization actions.

It can be conclude from Table 1 that the geometric mean of simulation slowdown compared to native execution for 1,2,4,8 threads are 55.7%, 91.8%, 105%, and 155% respectively, and an overall slowdown of 98.9% for all the cases, i.e., the simulation time is only doubled as compared with native execution time. Whereas other full-system architectural simulators typically has 100-1000× slowdown.

To conclude, DecompositionJ offers an intermediate modeling and simulation approach between the macroscopic Grid and Peer-to-Peer simulators, and the microscopic architectural simulators. It enables the modeling of concurrent Java executions with higher functional and timing fidelity then Grid and Peer-to-Peer simulators, and achieves lower overheads as compared with full-system architectural simulators.
VIII. CONCLUSION AND FUTURE WORK
This paper presents the DecompositionJ framework, which is a new approach to deterministically model and simulate concurrent data-race-free Java execution while exploiting host parallelism. Determinism is achieved by generating a total order over all actions, and parallelism is exploited by enforcing a relaxed clock-synchronization order on synchronization actions only. The proposed implementation is compatible with existing Java execution environments and tools since it does not require (i) additions to the language, e.g., special language directives; (ii) new hardware support; and (iii) programmer annotation. Simulation performance has been evaluated on the Parallel Java Grande Benchmark suite. Evaluation results have shown that the geometric mean overhead of DecompositionJ is only 98.9% as compared with native executions, which significantly outperforms the existing full-system simulation methods. It is believed that the high-efficiency, high-compatibility, and determinism properties of DecompositionJ are important for enabling simulation studies of a wide range of cyber-physical systems. Future development of the framework will focus on the following aspects:

i) Expand the simulation model to include compare-and-swap and thread parking operations, such that advanced synchronization mechanisms can be supported efficiently.

ii) Reduce simulation overhead by avoiding heavy-weight logical time barriers. It is recognized that most variable accesses are not in conflict: repeated accesses of variable by the same thread, or read-only accesses by multiple threads. These non-conflicting accesses does not create cross-thread dependency, hence need not be constrained by logical time barrier. Future development aims to detect non-conflicting accesses during runtime, and reduce redundant synchronization overhead.

iii) In this paper, the temporal model (i.e., $\tau$) of a target program is not discussed in detail. This is because performance modeling of a virtual machine based execution is very complicated, execution time can be affected by multiple factors transparent to the target program: (1) runtime profiling and recompilation, (2) garbage collection, (3) variations in virtual machine implementation. Further investigation is needed to create accurate timing models.

REFERENCES
WING-HONG LAU (M’88–SM’06) received the B.Sc. and Ph.D. degrees in electrical and electronic engineering from the University of Portsmouth, Portsmouth, U.K., in 1985 and 1989, respectively. He joined the City University of Hong Kong, where he is currently an Associate Professor with the Department of Electronic Engineering in 1990. His current research interests include digital signal processing, digital audio engineering, pulse width modulation spectrum analysis, embedded system design, and smart-grid development. Dr. Lau received the IEEE Third Millennium Medal. He was the Chairman of the IEEE Hong Kong Section in 2005.

TIAN MAO (S’14) received the B.S. degree in electrical engineering and its automation and the M.S. degree in electrical engineering from Hunan University in 2010 and 2013, respectively, and the Ph.D. degree from the City University of Hong Kong in 2017. He is currently with the Electric Power Research Institute, China Southern Power Grid. His research interests include power system operation, electric vehicle charging scheduling, and smart-grid energy management and optimization.

HENRY SHU-HUNG CHUNG (M’95–SM’03–F’16) received the B.Eng. and Ph.D. degrees in electrical engineering from The Hong Kong Polytechnic University in 1991 and 1994, respectively. Since 1995, he has been with the City University of Hong Kong. He is currently a Professor with the Department of Electronic Engineering and the Director of the Centre for Smart Energy Conversion and Utilization Research. His research interests include renewable energy conversion technologies, lighting technologies, smart-grid technologies, and computational intelligence for power electronic systems. He has edited one book, and he has authored eight research book chapters and over 355 technical papers, including 178 refereed journal papers in his research areas, and holds 42 patents. Dr. Chung has received numerous industrial awards for his invented energy saving technologies. He was the Chair of the Technical Committee of the High-Performance and Emerging Technologies, IEEE Power Electronics Society, from 2010 to 2014. He is currently the Editor-in-Chief of the IEEE POWER ELECTRONICS LETTERS and an Associate Editor of the IEEE TRANSACTIONS ON POWER ELECTRONICS and the IEEE JOURNAL OF EMERGING AND SELECTED TOPICS IN POWER ELECTRONICS.

NORMAN CHUNG-FAI TSE (M’09) received the B.Sc. degree from The Hong Kong Polytechnic University (then Hong Kong Polytechnic), Hong Kong, in 1985, holding an Associateship in electrical engineering, the M.Sc. degree from the University of Warwick, Coventry, U.K., in 1994, and the Ph.D. degree from the City, University of London, London, U.K., in 2007. He is currently with the Centre for Smart Energy Conversion and Utilization Research, City University of Hong Kong. His current research interests include power quality measurement and analysis, Web-based power quality monitoring, harmonics mitigation, and building energy efficiency study.

KIM-FUNG TSANG (M’95–SM’15) received the Associate degree in electrical engineering from The Hong Kong Polytechnic University in 1983 and the M.Eng. (by research) and Ph.D. degrees in electrical engineering from the University of Wales College of Cardiff (formerly known as the University of Wales Institute of Science and Technology), Cardiff, U.K., in 1987 and 1995, respectively. He joined the City University of Hong Kong in 1988, where he is currently an Associate Professor with the Department of Electronic Engineering. He has published about 200 technical papers and four books/chapters. Dr. Tsang is a fellow of HKIE, a Chartered Engineer and a member of IET, an Associate Editor and a Guest Editor of the IEEE TRANSACTIONS ON INDUSTRIAL INFORMATICS, an Associate Editor of IEEE Industrial Electronics Magazine and IEEE ITeN, and an Editor of the KSII Transactions on Internet and Information Systems.

LOI LEI LAI (SM’92–F’07) received the B.Sc. (First Class Hons.), Ph.D., and D.Sc. degrees from Aston University and the City University of London, respectively. He was the Director of the Research and Development Centre, State Grid Energy Research Institute, China; a Pao Yue Kong Chair Professor with Zhejiang University, China; a Guest Professor with Fudan University, China; a Vice President for the IEEE Systems, Man and Cybernetics Society; a Professor and a Chair in electrical engineering with the City University of London; and a Fellow Committee Evaluator for the IEEE Industrial Electronics Society. He is currently a University Distinguished Professor with the Guangdong University of Technology, Guangzhou, China. He is a fellow of the IET, a National Distinguished Expert in China, and a Distinguished Expert in State Grid Corporation of China. He was a recipient of the IEEE Third Millennium Medal, the IEEE Power and Energy Society UKRI Power Chapter Outstanding Engineer Award in 2000, the IEEE/PES Energy Development and Power Generation Committee Prize Paper in 2006 and 2009, the IEEE/SMCS Outstanding Contribution Award in 2013 and 2014, and the Most Active Technical Committee Award in 2016.

***