Asia-Pacific Journal of Information Technology and Multimedia 14(1) 2025: 351 – 366
https://doi.org/10.17576/apjitm-2025-1401-20

Software Model Checking for Distributed Applications using Hybridization of Centralization and Cache Approaches

Penyemakan Model Perisian bagi Aplikasi Teragih menggunakan Pendekatan Hibrid Pemusatan dan Cache

Hing Ratana[1], Sharifah Mashita Syed-Mohamad[2], Chan Huah Yong[1]*

[1] *School of Computer Sciences, Universiti Sains Malaysia, 11800 USM Penang, Malaysia*

[2] *Faculty of Computer Science and Mathematics, Universiti Malaysia Terengganu, 21030 Kuala Nerus, Terengganu, Malaysia Faculty of Computer Science and Mathematics, Universiti Malaysia Terengganu, 21030 Kuala Nerus, Terengganu, Malaysia*

**Corresponding author: s.mashita@umt.edu.my*

Received 17 September 2024
Accepted 23 May 2025, Available online 30 June 2025

ABSTRACT

Developing reliable distributed systems poses significant challenges due to the non-deterministic nature of thread and process execution, as well as communication channels. Software model checking offers a means to verify system correctness by exhaustively analyzing all program execution paths. However, the existing bytecode model checker, capable of verifying multiple processes, suffers from computational overhead. This paper introduces Java PathFinder (JPF)-Nas-Hybrid (JNH), a novel model checker addressing these limitations. JNH employs a redesigned inter-process communication (IPC) model and integrates a scalable caching mechanism. The experimental results show that the hybridization of centralization with cache significantly reduces the computational overhead and improves verification performance as well. Additionally, the paper explores bug detection strategies, distinguishing between local and global bugs, and evaluates various search strategies to explore distributed program state spaces. In every case, the proposed method results in a smaller state space, fewer bytecode instructions, and a shallower search graph.

Keywords**:** Distributed systems, Software model checking, Java PathFinder, Centralization approach, Cache-based approach

ABSTRAK

Membangunkan sistem teragih yang boleh dipercayai menimbulkan cabaran yang ketara disebabkan oleh sifat bukan penentu bagi pelaksanaan *thread* dan proses, serta saluran komunikasi. Pemeriksaan model perisian menawarkan cara untuk mengesahkan ketepatan

sistem dengan menganalisis secara menyeluruh semua laluan pelaksanaan program. Walau bagaimanapun, penyemak model kod bait sedia ada, yang mampu mengesahkan pelbagai proses, mengalami overhed pengiraan. Artikel ini memperkenalkan Java PathFinder (JPF)-Nas-Hybrid (JNH), penyemak model baru yang menangani batasan ini. JNH menggunakan model komunikasi antara proses (IPC) yang direka bentuk semula dan menyepadukan mekanisme caching berskala. Keputusan eksperimen menunjukkan bahawa penghibridan pemusatan dengan *cache* dengan ketara mengurangkan overhed pengiraan dan juga meningkatkan prestasi pengesahan. Selain itu, artikel ini meneroka strategi pengesanan pepijat, membezakan antara pepijat tempatan dan global, dan menilai pelbagai strategi carian untuk meneroka ruang keadaan program yang diedarkan. Dalam setiap kes, kaedah yang dicadangkan menghasilkan ruang keadaan yang lebih kecil, arahan kod bait yang lebih sedikit dan graf carian yang lebih cetek.

Kata kunci**:** Sistem teragih; Semakan model perisian; Java PathFinder; Pendekatan pemusatan; Pendekatan berasaskan cache

INTRODUCTION

Modern society depends heavily on complex software systems, which are integral to various sectors such as banking, automotive, retail, and entertainment. These systems are typically large, distributed, and designed to meet specific quality standards. As our reliance on these intricate distributed systems grows, ensuring their correctness and subjecting them to thorough analysis becomes crucial. The most common method for analyzing distributed systems is testing. After a system is designed, it is evaluated using a finite set of test cases to confirm it functions as intended. Although testing can be effective, especially when test cases are selected with relevant domain expertise, it is important to recognize that testing cannot guarantee success for all potential system behaviors.

On the other end of the analytical spectrum, formal methods use mathematical techniques to verify if a system meets a specified property under all possible conditions. One important sub-discipline of formal methods is model checking (Clarke et al. 2018). Conventional model checkers for distributed systems necessitate the utilization of abstract modeling languages like TLA+ (Lamport 1994), PlusCal (Lamport 2009), Coq (Barras et al. 1999), and SPIN (Holzmann 1997). This approach demands a substantial investment of developer effort and does not guarantee the identification of all bugs in the system implementation. These model checkers can only detect bugs within the specified system model and have no way of finding bugs in the actual implementation (Anand 2020).

A novel approach within the research community involves applying model checking directly to the actual implementations of distributed systems. The direct verification of real-world implementations enhances confidence in meeting software safety requirements. It's essential to recognize that adherence to system design specifications does not ensure compliance in the implementation phase. Numerous bugs related to concurrency, including race conditions, deadlocks, and assert violations, often come from programming errors during implementation. Verification during the design phase cannot definitively ensure the final deliverables will be free of bugs. Concrete model checkers (Yabandeh et al. 2009; Musuvathi et al. 2008; Lukman et al. 2019; Leesatapornwongsa et al. 2014; Laroussinie and Larsen 1998; Killian et al. 2007; Guo et al. 2011; Guerraoui and Yabandeh 2011; Deligiannis et al. 2016; Artho et al. 2017; Yang et al. 2009; Anand 2020; 2018) focus on testing and debugging unmodified distributed systems to detect failures, crashes, and violations of user-defined properties. They are model

checker tools for distributed systems, which are designed to address specific programming languages, such as Go (Anand 2020), or to examine the systems operating at the operating system level to identify bugs. Furthermore, concrete model checkers suffer from massive state space explosion and programming language coverage.

Unlike any other concrete model checkers, the Java Pathfinder (Artho et al. 2024) or JPF model checks bytecode rather than native code or operating system code. This approach reduces a large number of unrelated state spaces. Additionally, JPF offers fundamental support for verifying distributed systems (Artho and Visser 2019). It provides a structure for examining Java bytecode, primarily utilizing an explicit state model checker. Created by the Robust Software Engineering Group at NASA Ames Research Center, JPF has been available as open source since 2005.

Two primary approaches have been utilized for model-checking distributed systems during bytecode-level executions: centralization (Shafiei and Mehlitz 2014) and caching (Artho et al. 2009). Centralization involves capturing multiple processes and inspecting them for both local and global faults. This method requires Inter-Process Communication (IPC) for storing communication data. A tool for centralization has been developed as an extension to JPF, referred to as JPF-NAS (Network Asynchronous System). Conversely, the caching approach captures only one process at a time, allowing other processes to function within their native environments. This approach seeks to minimize the state space by analyzing one process at a time and is implemented as NET-IOCACHE (Network Input/Output Cache), an enhancement of JPF. Communication data in this technique is stored in a branching-time cache (Artho et al. 2009), storing either the server or client side of communication data.
The current JPF-NAS tool uses two ArrayByteQueue buffers—one for the server to send data to the client and another for the client to send data to the server. These buffers store communication data and process them byte by byte. During state exploration, the centralization process writes one character to the queue, moves to the next state, removes the same character from the queue, and updates the state accordingly. Unfortunately, the write and read operations (which write and remove data from the queue) of the data streams result in computational overhead.

In contemporary distributed systems, bytecode-level executions pose significant computational challenges, especially within centralized approaches. Our research aims to explore these challenges, seeking solutions to enhance system efficiency and performance. Specifically, we focus on addressing the intricate computational hurdles encountered by distributed systems, with a particular emphasis on bytecode-level executions within centralized frameworks. Our central inquiry revolves around how a hybridized approach, integrating caching mechanisms at the model checker level within centralized systems, can effectively manage computation overhead, mitigate the state space explosion problem, and optimize communication data management between processes.

This paper is structured as follows: Section 2 covers the background, detailing the formalism for defining distributed systems and the design of a scalable branching-time cache, along with its integration into centralization. Section 3 outlines our methodology. Section 4 presents the results and discussions. Finally, Section 5 concludes and suggests future research directions.

BACKGROUND

Labeled Transition Systems with Inputs and Outputs. A labelled transition system featuring inputs and outputs serves as a mathematical construct employed to precisely articulate the functioning of communication systems (Leungwattanakit et al., 2014; Bos, P. V. D., & Vaandrager, F. 2021). . The behaviour of this system can be represented by a labelled transition system, which is a 5-tuple $<S, L_I, L_O, T, s_0>$, where:

- S is a finite set of process states;
- L_I and L_O are a set of input labels and a set of output labels, respectively;
- $L_I \cap L_O = \emptyset$.
- $T \subseteq S \times (L_I \cup L_O \cup \{\tau\}) \times S$ is a set of transitions;
- $\tau \notin L_I \cup L_O$ is an unobservable action;
- $s_0 \in S$ is the initial state;

A process transitions from one state to another state based on a label. This label signifies an action within the process, which can be an input from another process, an output to another process, or an action that is not externally observable. The unobservable action can be viewed as an internal computation that occurs without interaction with the external environment. While there may be multiple distinct unobservable actions, this formalization simplifies by using the label τ to represent all of them without differentiation. In contrast to unobservable actions, the action $l \in L_I \cup L_O$ is observable. The set of labels for the LTS of P, represented as $L(P)$, includes every label, that is $L_I \cup L_O \cup \{\tau\}$.

Consider a distributed system made up of a single process that includes a finite number of threads represented by the set Φ. Each thread T in Φ performs a series of actions. The behaviour of this system can be represented by a transition system (TS), which is a tuple $<S, Act, \rightarrow, s_0>$, where:

- S is a set of states,
- Act is a set of actions,
- $\rightarrow \subseteq S \times Act \times \Phi \times S$ is the transition relation, and
- $s_0 \in S$ is the initial state.

The set of actions Act is divided into the set of visible actions V, and the set of invisible actions I. Instead of $(s, \alpha, T, s') \in \rightarrow$, it is written as $s \xrightarrow[T]{\alpha} s'$. The action α is said to be enabled in the state s, if

$$\exists T \in \Phi : \exists s' \in S : s \xrightarrow[T]{\alpha} s'$$

Statement 2.2 describes if there exists a thread T in a set of threads Φ, then there exists next state s' in the set of states S. Therefore, the state s moves to the state s' for each thread T taking an action α.

It uses enable(s) to denote the set of all enabled actions of any state s, that is,

$$enabled(s) = \{\alpha \in Act \mid \exists T \in \Phi: \exists s' \in S: s \xrightarrow[T]{\alpha} s'\}$$

Statement 2.3 describes the set of all enabled actions of any state s that is if there exists an action α in a set of actions Act or an element of thread T in the set of threads Φ, then the next state s' exists in the states S; therefore, the thread T moves from the state s to the next state s' by taking action α.

Each thread is assumed to be deterministic. At any state s, for each thread T, there is at most one action α and state s', where $s \xrightarrow[T]{\alpha} s'$. This is expressed by

$$if\ s \xrightarrow[T]{\alpha'} s'\ and\ s \xrightarrow[T]{\alpha''} s''\ then\ \alpha' = \alpha''and\ s' = s''$$

Think about a distributed system made up of a collection of (multi-threaded) processes. Let the set of threads Φ in the system represent all threads regardless of the process they belong to. The behaviour of this system can be represented by modeling it as a single process system made up of these threads and using a transition system TS.

A state s is called a global state if,

$$enabled(s) \subseteq V$$

The set of global states is denoted by g(S). Assuming that all actions to be executed from the initial state are visible and expressed by

$$\text{enabled}(s_0) \subseteq V$$

As a consequence, $s_0 \in g(S)$.

It's believed that the actions of one thread that are not visible do not impact actions taken by other threads. In software model checking, the focus is on the system states and transitions, which are influenced by instruction execution, including interactions with other systems. A state in a software system can consist of stack and heap data, as well as information about threads.

Stream Abstraction. A request is a message sent from a process that is being analyzed using a model checker to an external process. The response is the message received by the process from the external process. A data stream s is a finite sequence of messages $m: s = < m_0, \dots, m_{|s|-1} >$. A stream pointer $sp_s = i$ refers to a specific index i in the message sequence of a given data stream. A communication trace $t = < req, resp, limit >$ consists of two data streams, a request stream and a response stream, and a limit function $limit(sp_{req}) : sp_{resp}$ that maps a request pointer to its corresponding response pointer.

A program works with a set of communication traces t (which are equivalent to streams or sockets in a particular programming language). Each socket is linked to a single communication trace. The standard program state consists of a global heap and multiple threads, each with its program counter and stack, as well as a pair of stream pointers for every communication trace. This expanded program state is monitored by the model checker and is subject to backtracking. All communication traces must be consistent with the first trace observed. In any possible program execution, there must be a single trace t' such that, for all thread schedules, t = t' when the program ends normally.

Properties of Verification Systems. Errors in an LTS represent unfavorable conditions within a system, encompassing issues like local and global deadlocks, assert violations, and unhandled exceptions. Model checking aims to identify and locate these error states. Model checkers provide a conclusive determination regarding the existence of system faults through an exhaustive exploration of the state space.

The Existing Caching Techniques: The study by (Artho et al. 2009) proposes the concept of caching input/output (I/O) traces for verifying various network applications. They introduce a tree data structure called branching-time cache, which accommodates diverging

communication traces between different thread schedules. In this approach, communication traces are captured and stored in a tree, as depicted in Figure 1. Figure 1 illustrates the branching-time cache data structure for two cache entries. The request pointer moves from one node to another. Non-root nodes can be either requesting nodes or response nodes. Requesting nodes can have either one or more request child nodes or one response node, but not both. Response nodes can only have one or more request child nodes. Each response node can contain multiple characters.

With the branching cache data structure, a cache entry can hold a group of request nodes and a response node. However, this technique requires restarting the peer to generate the corresponding response for the request. The revised version of the I/O cache approach introduces a data structure known as the request/response tree (abbreviated as the RR tree). The RR tree is a tree data structure that always includes the root node, making it non-empty. Every descendant of the root node has a positive integer identification (ID) and an event node other than the root node is categorized as either a requesting node or response node based on the type of event, which is also grouped.

The ID of the root node is always zero. Events are classified as either request events or response events. A request event represents a communication event initiated by the target program, such as sending a character, while a response event represents an event initiated by the external environment that affects the target program, such as accepting a connection request or receiving a character. Request events and response events are contained in request nodes and response nodes, respectively, which are attached to the RR tree. Request nodes can be followed by either request nodes or response nodes. The tree also includes pairs of state pointers, request pointers, and response pointers. The request pointer points to the node representing the most recent message sent by the target program, while the response pointer points to the node representing the most recent message read by the target program.

In the case of multiple logical connections, there can be more than one pair of request and response pointers in the tree. Each pair of pointers corresponds to one logical connection, and the number of pairs equals the number of logical connections referenced by the target program. All connections share the same tree, assuming that they connect the target program to the same peer. If the target program happens to connect to multiple peers, the cache layer would create a separate tree for each peer. Although this work does not cover scenarios with multiple peers, it is a potential area of future support.

For each pair of pointers, the request pointer and the response pointer always exist on the same trace. Not all response messages in the tree are available. A response that has not been requested should not be visible to the target process. Only one pair of request and response pointers is active at any given time to simplify operations on the RR tree. It is referred to the node pointed to by the active request/response pointer as the active request/response node.

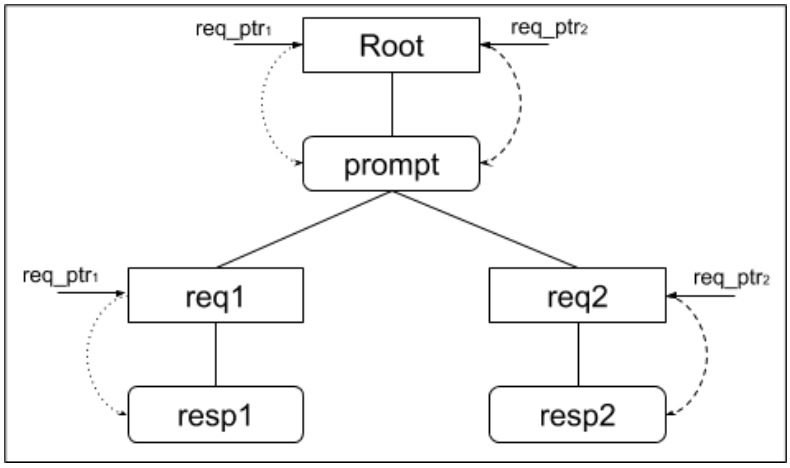

FIGURE 1. Branching-time request/response tree.

METHODOLOGY

The research methodology involves three main experimental steps: preliminary, experimental, and evaluation. In the preliminary phase, experiments on centralization and caching are conducted, and limitations are identified. Initially, experiments are conducted on centralization and caching using benchmark distributed applications, and information logs are gathered. Subsequently, the study identifies the limitations of centralization by meticulously examining specific areas for improvement.

Based on experimental findings, the study leans towards investigating the branching-time cache due to its adaptable design and primary role in storing communication data for model checking in distributed systems. Considering that the branching-time cache only stores one side of communication data, either client or server data, the study proposes a scalable branching-time cache capable of storing communication data from multiple processes. Ultimately, the research advocates for a hybrid model checker designed to integrate with the previously designed branching-time cache.

The second part of the research methodology entails the experimental process, which includes comparing existing centralization with the proposed centralization. Both model checkers are evaluated in terms of computational overhead. Finally, the concluding phase involves assessing the proposed work by evaluating both model checkers based on computational overhead using a JVM monitor. Figure 2 depicts the comprehensive research methodology employed in this study.

The Proposed Reduction Techniques. This section discusses the formal models and definitions essential for supporting the proposed hybridization of centralization and cache approaches. The branching-time cache has been modified for scalability and enabling it to store multiple process communication data rather than just individual process communication data. Subsequently, the endeavor integrates the scalable cache into the centralization model checker to address computational overhead and reduce the state space. Two significant modifications to the current limitation of the existing Inter-Process Communication (IPC) are outlined as follows:

1. With the communication data now accessible in the cache, the newly designed IPC employs multiple pointers to navigate through the data. This approach is significantly

more cost-effective than the previous method of writing and reading (read and remove) one byte at a time with ArrayByteQueue during backtracking, thereby minimizing computational overhead.

2. Instead of processing communication data byte by byte during the backtracking process, the updated IPC now processes data in multi-byte chunks, thereby reducing the number of explored states.

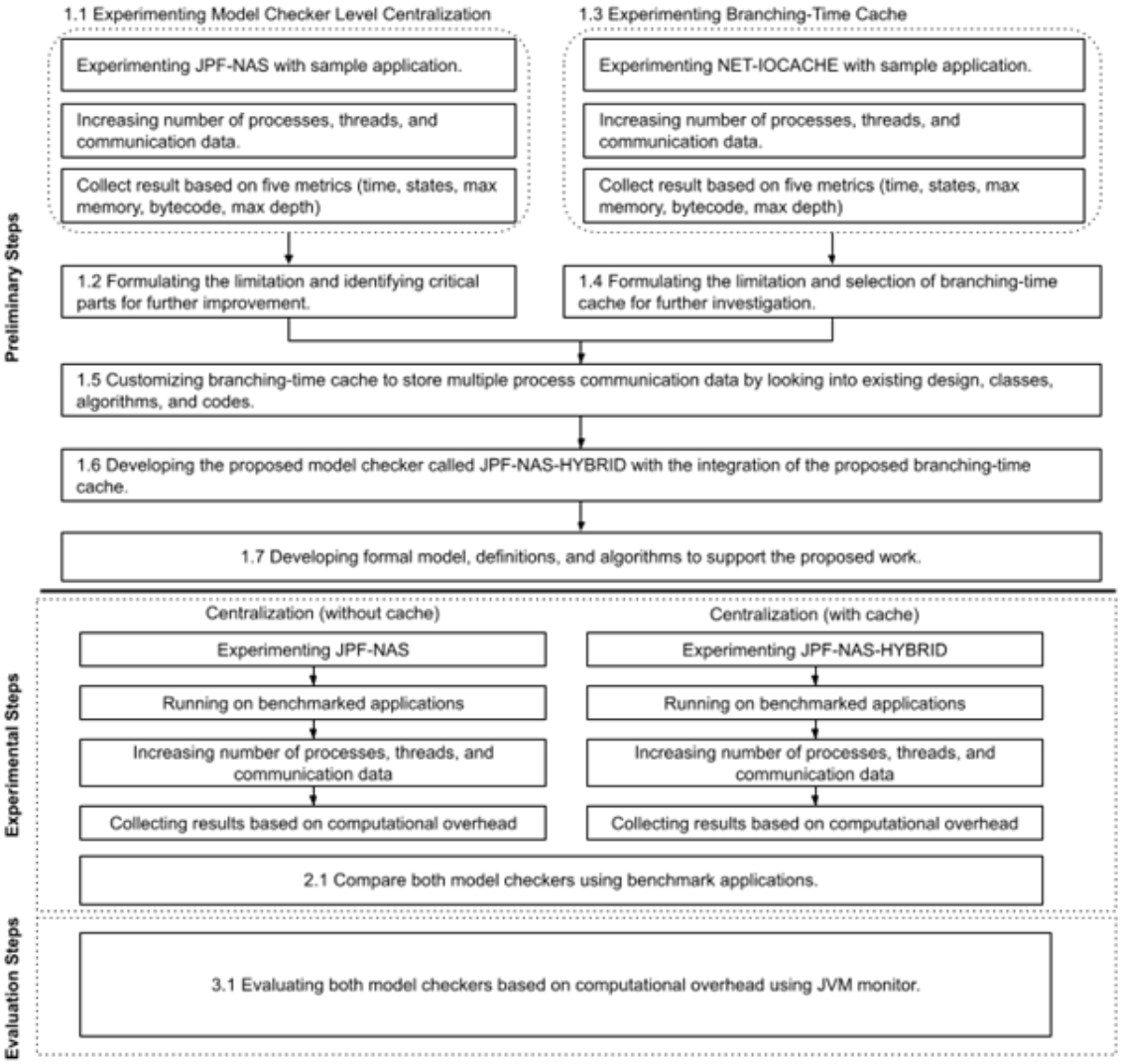

FIGURE 2. Research methodology framework

Computational overhead in model checking refers to the additional computational resources required to perform the verification or analysis of a system when compared to its baseline, unmodified form. Consider a cache C is a set of cached data, where each cached state c ∈ C is associated with a unique identifier and stores the result of some previously performed computation.

Definition 1: A state space exploration is a sequence of states $s_0, s_1, \ldots, s_n$ such that for each i, $0 \leq i < n$, there exists an action $a_i \in Act$ such that as $s_i \rightarrow s_i+1$ by taking action a_i.

Definition 2: Given a state *s* in the exploration, if *s* is found in the cache *C*, it is a cache hit. Otherwise, it is a cache miss.

Definition 3: The cache utilizes pointers to efficiently traverse through the data stored in memory. Each cached state c may have pointers that indicate connections to other cached states or relevant information.

Let *E* be the total computational cost of using "ArrayByteQueue", and *E′* be the total computational cost with the applied cache pointers. Computational overhead reduction *(R)* is formally defined as $(R) = E - E'$. This reduction cost quantifies the difference in computational overhead between the two approaches, indicating how much more efficient the multiple pointers in the cache are compared to the queue-based approach.

For example, suppose the model checker has a SUT where communication data needs to be frequently accessed and processed. Let's say the queue-based approach takes $E = 1000$ computational units for a specific workload, while the multiple pointers in cache take $E' = 500$ computational units for the same workload. The computational reduction cost would then be $(R) = 500$ computational units. This means that using multiple pointers in a cache reduces the computational overhead by 500 units compared to the queue-based approach for the given workload.

EVALUATION

Our study evaluated a set of distributed Java applications, detailed in Table 1. These programs, referenced in previous research (Shafiei and Mehlitz 2014), range from simple, single-threaded applications (Echo) to more complex, multi-threaded ones (Alphabet). Echo involves single-threaded server and client processes, whereas Alphabet includes multi-threaded server and client processes.

TABLE 1. Distributed Java Applications

Application	Size (LOC)	Min. Thread	Architecture
Echo server	25	1 main thread	Client/server
Echo client	38	1 main thread, 1 worker thread	Client/server
Daytime server	22	1 main thread	Client/server
Daytime client	34	1 main thread, 1 worker thread	Client/server
Alphabet server	40	1 main thread, 1 worker thread	Client/server
Alphabet producer	14	1 main thread	Client/server
Alphabet consumer	19	1 main thread	Client/server
Alphabet client	31	1 main thread	Client/server

The empirical analysis of these benchmark applications focused on various factors such as computational performance, time complexity, maximum memory consumption, state space exploration, bytecode execution, and search tree depth, as noted in prior studies (Leungwattanakit et al., 2014; Shafiei & Mehlitz, 2014). The following sections elaborate on the empirical analysis of the experiments.

Computational Performance and Time Complexity: We measure the algorithm's efficiency by evaluating how long it takes to execute and the computational resources it consumes. This includes analyzing the time complexity to understand the scalability of the proposed method with increasing input sizes.

Maximum Memory Consumption: The memory usage of the algorithm is monitored to ensure it remains within acceptable limits. This is crucial for applications with limited memory resources.

State Space Exploration: We leveraged Java PathFinder (JPF) to explore the state space using multiple listeners:

1. DistributedSimpleDot Listener: Generates a dot file that visualizes the search graph explored by JPF. It differentiates between local and global scheduling points, depicted as circles and octagons, respectively.
2. StateSpaceAnalyzer Listener: Collects data on the choices made during the model-checking process, allowing identification of factors contributing to the size of the state space.
3. JPF Logging System: Used for troubleshooting and understanding the execution of bytecodes on the JPF JVM level.

Hardware and Software Setup: Experiments were run on a machine with the following configuration:

Operating System: Windows 10 Pro 64-bit (10.0, Build 19044)
Processor: Intel(R) Core(TM) i7-10870H CPU @ 2.20GHz (16 CPUs, ~2.2GHz)
Memory: 16GB
Java Development Kit: 1.8.0_391
Java PathFinder Core System Version: 8.0
Eclipse IDE: 4.14.0
JVM Monitor Version: 3.8

This setup provided a reliable environment for testing and validating the method under realistic conditions.

RESULT AND DISCUSSION

A comparison of CPU usage between JPF-NAS and JNH for the Echo, Daytime, and Alphabet applications is presented in figures 3, 4, and 5 respectively. In these experiments, CPU usage was monitored over time, with the x-axis representing the computer time and the y-axis indicating CPU usage as a percentage.

During model checking with JPF-NAS, CPU usage remained consistently high. Notably, in the Echo application, there was a drop to 65%, but in most cases, the CPU maintained a 100% usage, leading to computational overhead. This sustained usage caused the CPU to halt, preventing it from handling other tasks efficiently.

On the other hand, JNH demonstrated a more balanced CPU usage. The system was able to release CPU resources as needed, allowing it to remain responsive to users and manage additional tasks concurrently. This performance difference highlights JNH's efficiency in reducing computational strain during model checking.

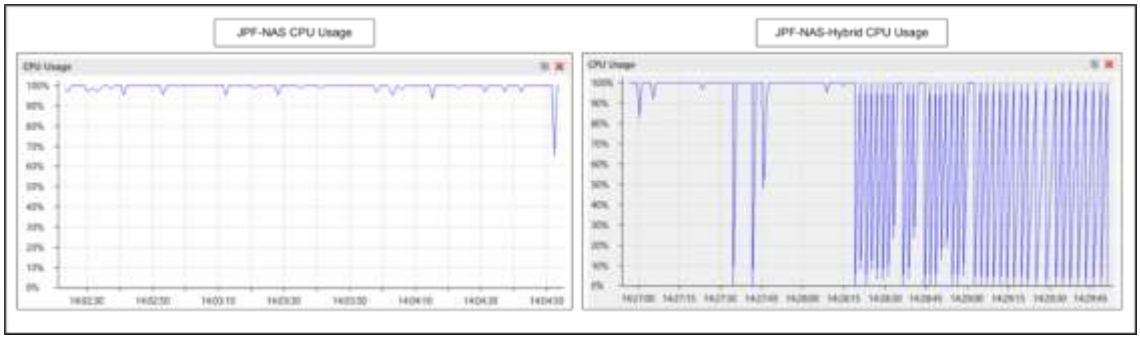

FIGURE 3. CPU Usage comparison between JPF-Nas and JNH for Echo application.

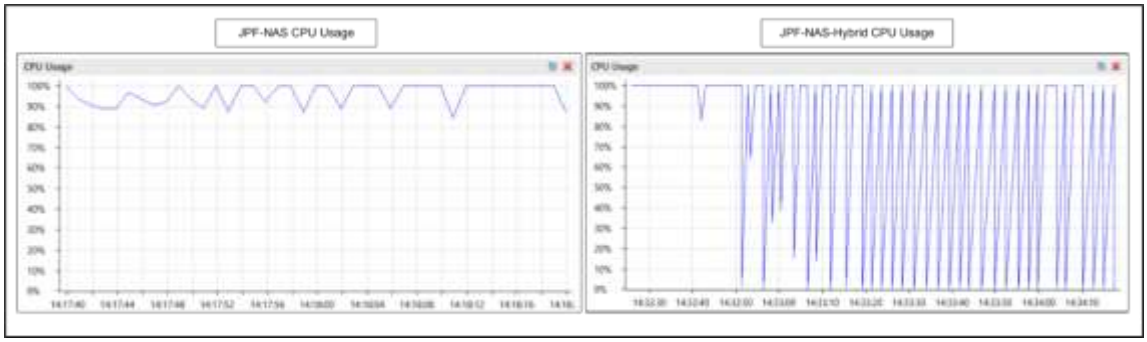

FIGURE 4. CPU Usage comparison between JPF-Nas and JNH for Daytime application.

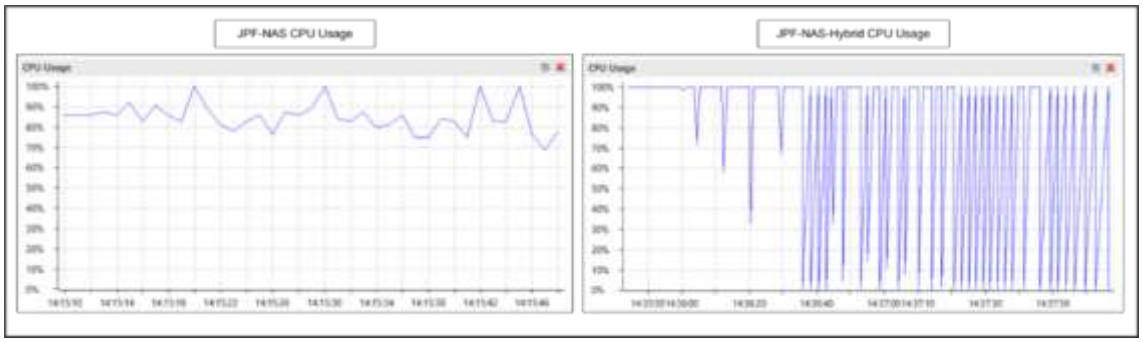

FIGURE 5. CPU Usage comparison between JPF-Nas and JNH for Alphabet application.

Tables 2, 3, and 4 present the execution time in seconds along with the number of states and maximum memory obtained from applying both approaches. The first column shows the increment of the two variables described above. For the experimental results, in every case, the proposed approach has better performance. Moreover, the number of states and the number of memory consumption does not increase exponentially; additionally, the proposed approach can model checks up to 5 clients and 5 servers while the existing centralization is not able to.

TABLE 2. Execution results obtained from model checking ***Echo*** application using JNH and JPF-NAS.

Thread		JPF-NAS-HYBRID			JPF-NAS		
Client	**Server**	**Time**	**States**	**Max Memory (MB)**	**Time**	**States**	**Max Memory (MB)**
1	1	0	60	243	0	66	243
2	2	0	741	243	0	1397	243
3	3	2	8767	432	8	27288	688
4	4	28	98544	432	158	483298	2416
5	5	411	1075796	782	N/A	N/A	N/A

TABLE 3. Execution results obtained from model checking ***Daytime*** application using JNH and JPF-NAS.

Thread		JPF-NAS-HYBRID			JPF-NAS		
Client	**Server**	**Time**	**States**	**Max Memory (MB)**	**Time**	**States**	**Max Memory (MB)**
1	1	0	62	243	0	100	243
2	2	7	9145	688	23	92598	991

TABLE 4. Execution results obtained from model checking ***Alphabet*** application using JNH and JPF-NAS.

Thread		JPF-NAS-HYBRID			JPF-NAS		
Client	**Server**	**Time**	**States**	**Max Memory (MB)**	**Time**	**States**	**Max Memory (MB)**
1	1	0	549	243	0	100	243
2	2	61	227387	687	82	301409	1691

Table 2 shows the execution results for model checking the Echo application, where performance metrics are provided for different configurations involving various numbers of threads and clients. The metrics for each configuration include the execution time (in seconds), the number of states explored, and the maximum memory usage (in megabytes). In the case of a single thread with one client and no server, JPF-NAS-HYBRID completed the task in 60 seconds, exploring 243 states, with a maximum memory usage of 243 MB. JPF-NAS, under the same configuration, took slightly longer at 66 seconds, exploring the same number of states with the same maximum memory usage.

As the number of threads and clients increased, the difference in performance between the two tools became more pronounced. With two threads and two clients, JPF-NAS-HYBRID completed the task in 741 seconds while maintaining the same number of states and memory usage as the single-thread case. JPF-NAS, however, took almost twice as long at 1397 seconds, though the states and memory usage remained the same.

For three threads with three clients and two servers, JPF-NAS-HYBRID took 8767 seconds, exploring 432 states with a maximum memory usage of 432 MB. In contrast, JPF-NAS required 27288 seconds to complete the task, exploring significantly more states (688) and using more memory (688 MB). The trend continues with higher thread counts. With four threads, four clients, and 28 servers, JPF-NAS-HYBRID completed the task in 98544 seconds, exploring 432 states with a memory usage of 432 MB. JPF-NAS, however, required a staggering 483298 seconds, exploring 2416 states and consuming 2416 MB of memory. For the highest configuration tested, with five threads, five clients, and 411 servers, JPF-NAS-

HYBRID completed the task in 1075796 seconds, exploring 782 states with a memory usage of 782 MB. Unfortunately, the data for JPF-NAS in this configuration is not available (N/A), indicating possible execution or resource limitations.

Table 3 provides the execution results for the Daytime application under different thread and client configurations. The table follows the same format as Table 2, listing the execution time, number of states explored, and maximum memory usage for each configuration. With one thread, one client, and no server, JPF-NAS-HYBRID completed the task in 62 seconds, exploring 243 states with a maximum memory usage of 243 MB. JPF-NAS, under the same conditions, took longer at 100 seconds, though the states explored and memory usage remained the same. When the configuration increased to two threads, two clients, and seven servers, JPF-NAS-HYBRID completed the task in 9145 seconds, exploring 688 states with a maximum memory usage of 688 MB. JPF-NAS, however, took significantly longer at 92598 seconds, exploring 991 states and using 991 MB of memory.

Table 4 details the execution results for the Alphabet application, with performance metrics provided for different configurations involving various numbers of threads and clients. For one thread, one client, and no server, JPF-NAS-HYBRID completed the task in 549 seconds, exploring 243 states with a maximum memory usage of 243 MB. JPF-NAS, under the same conditions, took only 100 seconds, exploring the same number of states with the same memory usage. With two threads, two clients, and 61 servers, JPF-NAS-HYBRID took 227387 seconds, exploring 687 states with a maximum memory usage of 687 MB. In contrast, JPF-NAS required 301409 seconds, exploring 1691 states and consuming 1691 MB of memory.

Bug Seeding. In distributed systems, different components of the system may need distinct software and hardware, and failures can happen at various levels. Identifying potential issues may require simulating failures across multiple layers. Previous findings indicate that the proposed JNH model checker can identify the global deadlock of the Echo application. In this section, the work introduces the injection of bugs in experimental distributed applications and demonstrates that the JNH model checker with the newly designed scalable cache retains its ability to detect local bugs, which depends on the local scheduler of the JPF core system.

The purpose of these experiments is to find out whether the proposed method still preserves local bug events when extending functionalities from the JPF core system. There are two parts in that the bugs are injected into the system under test (SUT). The first part is the injection of the codes right after the main method and the end of the main method. These are done for Echo and Daytime applications, shown in Figure 6. Another part of bug injection is performed at every start of the thread and the end of the threads. Therefore, these experiments will ensure that the local scheduler is working well during the model-checking process.

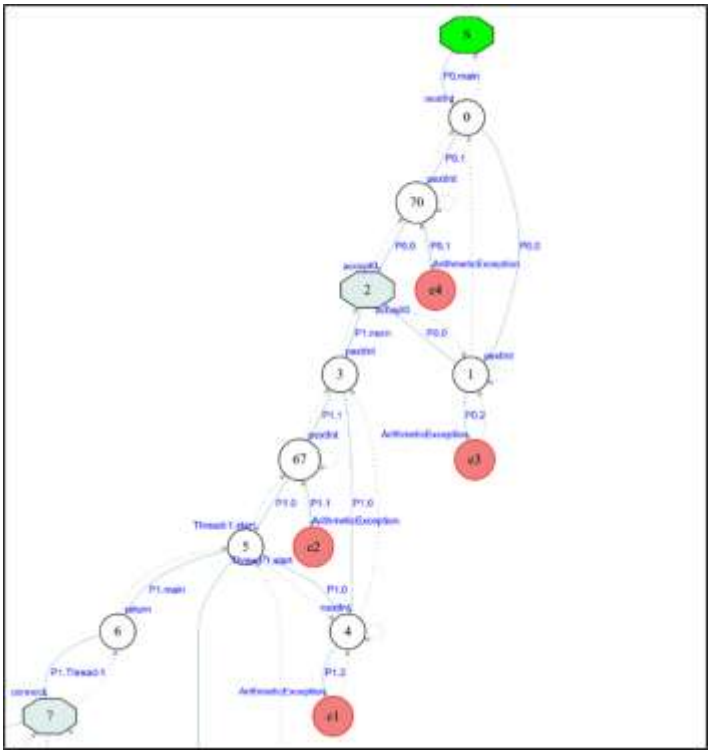

FIGURE 6. Example of ***Echo*** and ***Daytime*** bug seeding.

When the model-checking process starts, the execution will be aborted when there is an error found. To enable the JPF execution to continue running until completion, the property "search.multiple_errors" is set to true. Figure 7 illustrates the example of Chat and Alphabet bug seeding.

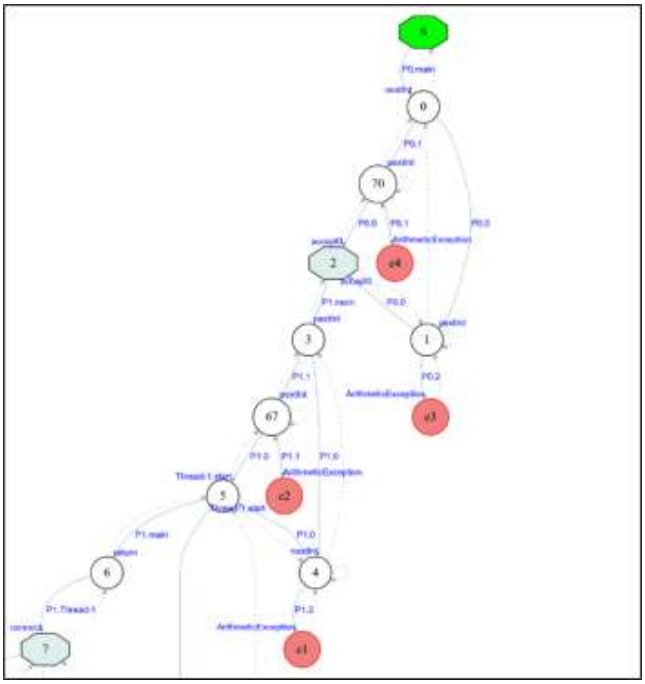

FIGURE 7. Example of ***Alphabet*** bug seeding.

CONCLUSION

The article discusses advancements in verifying distributed multithreaded Java applications, particularly emphasizing enhancements in model-checking methods at the bytecode execution stage. Prior strategies have implemented centralization and caching techniques to authenticate distributed systems, each possessing distinct advantages and drawbacks. This work opts to refine the centralization approach due to its capability to accommodate multiple processes. However, the existing inter-process communication (IPC) design within the centralization results in excessive computational overhead. The branching-time cache has been customized and the suggested alterations for managing inter-process communication, such as relocating the request and response tree and processing data in multi-byte chunks, showcase approaches for reducing computational overhead.

ACKNOWLEDGEMENT

This research was supported by the Talent and Publications Enhancement Research Grant (TAPERG/2023/UMT/2223), titled “Empirical Analysis of Software Maintainability Metrics in DevOps Environments” awarded by Universiti Malaysia Terengganu.

REFERENCES

Anand, V. 2018. Dara: Hybrid model checking of distributed systems. In *ESEC/FSE 2018 - Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (pp. 977–979). https://doi.org/10.1145/3236024.3275438.

Anand, V. 2020. *Dara the Explorer: Coverage based exploration for model checking of distributed systems in Go* (Master's thesis). University of British Columbia.

Artho, C., Quentin, G., Guillaume, R., Kazuaki, B., Lei, M., Takashi, K., Hagiya, M., Tanabe, Y., & Yamamoto, M. 2017. Model-based API testing of Apache Zookeeper. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)* (pp. 288–298).

Artho, C., Leungwattanakit, W., Hagiya, M., Tanabe, Y., & Yamamoto, M. 2009. Cache-based model checking of networked applications: From linear to branching time. In *ASE 2009 - 24th IEEE/ACM International Conference on Automated Software Engineering* (pp. 447–458). https://doi.org/10.1109/ASE.2009.43.

Artho, C., Păsăreanu, P., Quang, D., Gade, V., & Pu, Y. 2024. JPF: From 2003 to 2023. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (pp. 3–22).

Artho, C., & Visser, W. 2019. Java Pathfinder at SV-COMP 2019 (Competition contribution). *Lecture Notes in Computer Science, 11429,* 224–228. https://doi.org/10.1007/978-3-030-17502-3_18.

Barras, B., Boutillier, S., Cornet, C., Courant, J., Coscoy, Y., Delahaye, D., & Rémy, D. 1999. *The Coq Proof Assistant reference manual* (Version 6.11). INRIA.

Bos, P. V. D., & Vaandrager, F. 2021. State identification for labeled transition systems with inputs and outputs. *Science of Computer Programming, 209*, 102678.

Clarke, E. M., Henzinger, T. A., Veith, H., & Bloem, R. (Eds.). 2018. *Handbook of model checking* (Vol. 10). Springer.

Deligiannis, P., Matt, M., Thomson, P., Chen, S., Francis, A. D., Emmons, J., & Huang, C. 2016. Uncovering bugs in distributed storage systems during testing. In *14th USENIX Conference on File and Storage Technologies (FAST 16)* (pp. 249–262).

Guerraoui, R., & Yabandeh, M. 2011. Model checking a networked system without the network. *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*.

Guo, H., Wu, M., Zhou, L., Hu, G., Yang, J., & Zhang, L. 2011. Practical software model checking via dynamic interface reduction. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (pp. 265–278).

Holzmann, G. J. 1997. The model checker SPIN. *IEEE Transactions on Software Engineering, 23*(5), 279–295.

Killian, C., Anderson, J. W., Jhala, R., & Vahdat, A. 2007. Life, death, and the critical transition: Finding liveness bugs in systems code. In *Proceedings of the Symposium*.

Lamport, L. 1994. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems (TOPLAS), 16*(3), 872–923.

Lamport, L. 2009. The PlusCal algorithm language. In *International Colloquium on Theoretical Aspects of Computing* (pp. 36–60).

Laroussinie, F., & Larsen, K. G. 1998. CMC: A tool for compositional model-checking of real-time systems. In *International Conference on Protocol Specification, Testing and Verification* (pp. 439–456).

Leesatapornwongsa, T., Hao, M., Joshi, P., Lukman, J. F., & Gunawi, H. S. 2014. SAMC: Semantic-aware model checking for fast discovery of deep bugs in cloud systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (pp. 399–414).

Leungwattanakit, W., Artho, C., Hagiya, M., Tanabe, Y., Yamamoto, M., & Takahashi, K. 2014. Modular software model checking for distributed systems. *IEEE Transactions on Software Engineering, 40*(5), 483–501. https://doi.org/10.1109/TSE.2013.49.

Lukman, J. F., Ke, H., Stuardo, C. A., Suminto, R. O., Kurniawan, D. H., Simon, D., Priambada, S., et al. 2019. FlyMC: Highly scalable testing of complex interleavings in distributed systems. In *Proceedings of the Fourteenth EuroSys Conference 2019* (pp. 1–16).

Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P. A., & Neamtiu, I. 2008. Finding and reproducing Heisenbugs in concurrent programs. In *OSDI* (Vol. 8).

Shafiei, N., & Mehlitz, P. 2014. Extending JPF to verify distributed systems. *ACM SIGSOFT Software Engineering Notes, 39*(1), 1–5. https://doi.org/10.1145/2557833.2560577.

Yabandeh, M., Knezevic, N., Kostic, D., & Kuncak, V. 2009. CrystalBall: Predicting and preventing inconsistencies in deployed distributed systems. In *The 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI'09)*.

Yang, J., Chen, T., Wu, M., Xu, Z., Liu, X., Lin, H., Yang, M., Long, F., Zhang, L., & Zhou, L. 2009. MODIST: Transparent model checking of unmodified distributed systems. In *NSDI'09* (pp. 213–228).