# Model Checking Distributed Systems by Combining Caching and Process Checkpointing

Watcharin Leungwattanakit*, Cyrille Artho†, Masami Hagiya*, Yoshinori Tanabe‡ and Mitsuharu Yamamoto§
*Graduate School of Information Science and Technology
The University of Tokyo, Tokyo, Japan
Email: {watcharin, hagiya}@is.s.u-tokyo.ac.jp
†Research Center for Information Security, AIST, Tsukuba, Japan
Email: c.artho@aist.go.jp
‡National Institute of Informatics, Tokyo, Japan
Email: y-tanabe@nii.ac.jp
§Chiba University, Chiba, Japan
Email: mituharu@math.s.chiba-u.ac.jp

***Abstract*—Verification of distributed software systems by model checking is not a straightforward task due to inter-process communication. Many software model checkers only explore the state space of a single multi-threaded process. Recent work has proposed a technique that applies a cache to capture communication between the main process and its peers, and allows the model checker to complete state-space exploration. Although previous work handles non-deterministic output in the main process, any peer program is required to produce deterministic output.**

**This paper introduces a process checkpointing tool. The combination of caching and process checkpointing makes it possible to handle nondeterminism on both sides of communication. Peer states are saved as checkpoints and restored when the model checker backtracks and produces a request not available in the cache. We also introduce the concept of strategies to control the creation of checkpoints and the overhead caused by the checkpointing tool.**

***Index Terms*—software model checking; caching; software verification; distributed systems; checkpointing;**

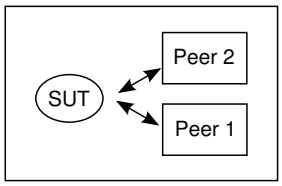
## I. Introduction

Networked software is usually implemented as a concurrent program using multiple *threads* to handle connections. Threads are execution units within a given process [1]. The interleaving among threads, i.e. thread scheduling, is taken care of by an operating system, thus it is beyond the control of programmers. As a result, *software testing* [2] may miss some failures under a certain sequence of interleaving, because it cannot cover all possible thread schedules in one run. Chess [3] remedies this disadvantage by executing a test case repeatedly to find concurrent failures and ensuring that every run takes a different interleaving. More program behaviors are tested by this technique. *Model checking* [4] is a more powerful verification technique that takes every possible schedule into account. Some software model checkers such as *Java PathFinder (JPF)* [5] execute real application code at runtime and are applied in the implementation phase of a software development. In this paper, the main process to be verified is called the *system under test (SUT)*. The system under test is backtracked by a model checker during verification to analyze multiple outcomes of non-deterministic decisions, such as thread scheduling and variable input data. The combination of decisions increases exponentially over the number of instructions. As a result, the program state space is usually too large to be explored exhaustively within a reasonable amount of time. This limitation is called the *state explosion problem*, which is one of the fundamental problems for model checking. Partial order reduction [6] is a technique to relieve the state explosion by atomically executing a group of program instructions that do not affect any other threads. This method reduces the number of thread interleavings, and thus the size of the state space.

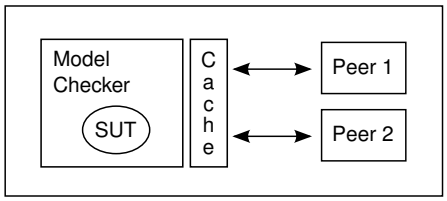
Verifying a distributed system [7] with a model checker is not a straightforward process. The distributed system is composed of several computational entities that exchange data and interoperate with one another through a network. Each process may run on a different environment, increasing system complexity. Most software model checkers only handle a single process at a time and cannot be applied simulatenously to all processes of the distributed system. When a process in the system is executed, as the SUT, by the model checker, the other processes are running as *peer processes* in the normal execution environment. Since the peer processes are not under model checker control, they cannot be backtracked in tandem with the SUT. After the SUT backtracks, it may try to interact with the peers, which are not in a state to respond correctly. Several techniques [8], [9], [10] have been established to automate the verification of such systems. Some of them are briefly introduced in Section II. Our previous work [11], [12] has shown that an *I/O cache* can interact with the SUT on behalf of the peer.

### A. Cache-based Verification

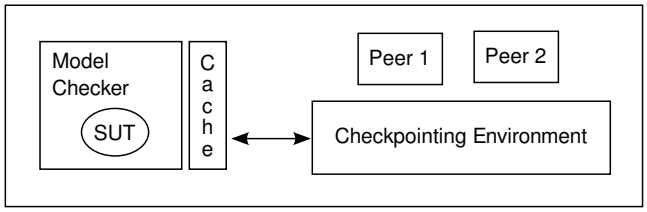
Fundamentally, *dynamic software verification* can be carried out by two approaches: testing and model checking. Figures 1a and 1b compare the configurations of both approaches in the verification of a distributed application. Testing executes both the SUT and peers in the normal execution environment. Only




(a) Software testing setup.




(b) Cache-based model checking setup.




(c) Checkpointing support setup.

Figure 1: Three configurations: testing, cache-based model checking, and model checking with checkpointing support.

one execution path of the SUT is exercised for each run in this configuration. On the other hand, model checking executes the SUT inside an environment where the program state may be rolled back. Thus, the SUT can be systematically driven through every possible execution path. In case of a multi-process application, the I/O cache is required to interact with the SUT on behalf of the peers [11], [12]. The I/O cache intercepts every *request packet*, a data packet sent by the SUT, and stores it in an internal data structure. Similarly, *response packets* coming back from the peers are stored in the I/O cache as well. Each request packet is matched with its corresponding response packet, if any. The I/O cache uses this information to imitate peer behaviors. As a result, the SUT experiences the same interaction with the I/O cache that would encounter with the actual peers. The single-process model checker then can complete the exploration of the SUT state space. In doing so, it avoids an expensive analysis of the full state space of each peer. Similar to a partial-order reduction, this reduces the state space significantly. By analyzing the full state space of the SUT combined with only a few (rather than all) peer executions, cache-based verification allows systems to be analyzed that were previously out of reach for model checking [11], [12].

In this research, determinism of programs is defined to be based on the output they produce with respect to input on a *communication channel*. Note that multithreaded programs whose thread schedules are non-deterministic can still produce deterministic output by this definition. We do not impose a restriction on "internal" non-determinism of programs. Furthermore, the term "deterministic output" means that the output solely depends on the input trace of the communication peer. A program with deterministic output may still produce a different output pattern if it receives a different input trace.

The initial implementation of the I/O cache assumed determinism of the SUT output [11]. However, this assumption is not always true. Some kinds of programs serve clients with dynamic data, e.g. web servers and database servers. Their outcomes do not only depend on the response from a peer but also on their internal state. Therefore, the I/O cache may observe multiple patterns of request packets from such programs running as SUT. We can say that the SUT behaves in a non-deterministic way from the perspective of the I/O cache. To handle programs with non-deterministic output, the I/O cache creates, for each distinct request pattern, a new instance of the peer. Each instance of the peer is responsible for one request pattern. While non-determinism on the SUT side is taken into account, previous work [12] assumes deterministic output from peers. Previous work restores a peer state by replaying previously recorded communication to a new instance of the peer [12]; non-deterministic peer systems cannot be handled in this way.

### *B. Extension for Non-deterministic Peers*

This paper proposes a method to support non-deterministic output from both SUT and peers with the help of *process checkpointing*. Process checkpointing is a technique that runs a group of processes in an environment that keeps track of the process states. This environment is called a *checkpointing environment*. Figure 1c shows the configuration of cache-based model checking with a checkpointing environment. The checkpointing environment creates a checkpoint of the peers when requested by the I/O cache. When the I/O cache needs to synchronize the state of the SUT and peers, the checkpointing environment restarts the peers from an appropriate checkpoint. This avoids replaying peer actions that may cause the previously executed non-deterministic transition to be repeated. Thus, the SUT only observes one peer behavior for each SUT output trace. This method eliminates false positives caused by different instances of peers interacting with the SUT under one execution path.

Nondeterminism inside a peer can be divided into two types by its source: thread scheduling and external input. Thread scheduling is controlled by an operating system. Even though a peer is loaded from a checkpoint, there is no guarantee that the peer will execute under the same thread schedule. Accordingly, we assume peer output of each communication channel is independent of thread scheduling.

Checkpointing a process is an expensive operation. Doing it naively would incur extremely high overhead. We propose strategies to prevent the model checker from creating unnecessary checkpoints. The contribution of this work is as follows:

- The application of process checkpointing to software model checking.
- Support for distributed applications that produce non-deterministic output.

- Introducing checkpointing strategies.
- A model checker extension that implements the proposed algorithm.[1]

### C. Outline

Section II shows related work to verify distributed applications. Section III presents how to make use of process checkpointing in software model checking. Section IV gives the implementation details of the model checker extension that supports peers with non-deterministic output. Section V presents and analyzes experimental results of the checkpoint-supported I/O cache on several systems under test. Section VI concludes the paper and proposes future work.

## II. Related Work

Several approaches have been presented to automate verification of distributed systems. The *Centralization* technique [8], [9] offers automatic unification of processes. It collects all processes in an application and transforms each process to a thread. All threads start inside a process called a *centralized process*, which can be automatically generated by a tool. A single-process model checker runs the centralized process, which starts all threads at the beginning, and verifies the entire system at once. Since all processes must be wrapped into one process, they must be written in the same programming language and be compiled on the same platform. These requirements are not always fulfilled. The centralization approach does not scale well since exploring the interleavings of all processes in the system yields very large state space.

Implementing a multi-process model checker is one of the solutions. This idea was proposed in [13]. The extension of *User-mode Linux* [14] called *ScrapBook* can save and restore the state of a system running inside a virtual environment. A SUT is executed inside the virtual environment. Note that there is no peer process in this approach, because every process is inside the virtual environment. Each process of the application is controlled by an instance of *GDB (GNU Debugger)* [15]. Given a set of breakpoints, GDB suspends the process. A user specifies these breakpoints beforehand. ScrapBook works as a model checker in the sense that it can save and revert the system state. Since the state of the entire system must be saved and restored during verification, this approach is not scalable as well.

*Verisoft* [16] is another model checking tool that verifies concurrent processes. It deals directly with the implementation of a target system, which may comprise multiple processes. However, it could not handle multi-threaded processes and did not maintain states of file descriptors for files and sockets that the system would open. Therefore, modern applications composed of multiple threads cannot be directly verified by the tool.

[1] http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/net-iocache

## III. Process Checkpointing Support

This section explains how process checkpointing is applied to verification of distributed systems. This idea is built on the concept of the I/O cache, which is reviewed in Section III-A. Some checkpointing tools are briefly introduced in Section III-B. We use process checkpointing to capture consistent behaviors of peer processes. Without process checkpointing, the I/O cache may store inconsistent data, which causes the model checker to report false positives. Such a situation is shown in Section III-C. We propose an optimization method in Section III-D in order to reduce the number of checkpoints and control the overhead caused by the checkpointing tool.

### A. Fundamentals of the I/O Cache

The I/O cache is a software module that controls data packets transferred between a SUT and a peer. It captures the messages sent by the SUT and matches them to the corresponding messages from the peer. A message from the SUT is called a *request message* while one from the peer is called a *response message*. A request message will be stored in an internal data structure if the I/O cache receives it for the first time. In this case, the I/O cache will poll the peer process for a response message. If a response message is available, the I/O cache will match it with the most-recent request message [11]. On the other hand, if the I/O cache receives a request message it has received before, it will send the response message associated with that request message to the SUT.

Figure 2 demonstrates how the I/O cache works on a two-thread SUT. Let $W$ and $R$ be threads that produce an output trace and receive an input trace, respectively. Thread $W$ randomly produces string '01' or '02'. In the first schedule, Thread $W$ writes '0', denoted by $W(0)$. The cache memorizes the data block transferred and shades the block to indicate that the SUT has passed through. Then, the I/O cache sends the request message to the peer and polls a response message. The response message is saved in the next block (Figure 2a). Note that the response message is not shaded, because the SUT has not read it yet. In the next step, thread $R$ attempts to read a message and receives the previously cached response message. The I/O cache shades the read block to mark that the SUT has already received this message (Figure 2b). Suppose that thread $W$ produces '1' in the next step, the I/O cache becomes like Figure 2c. When the SUT backtracks to state 2, the I/O cache restores the shade position, but the cached data remains permanent (Figure 2d). The model checker executes another possibility in which $W$ produces '2'. At this time, the peer is restarted from the beginning to handle the new request messages '02' (Figure 2e). The SUT backtracks to state 1. Thread $W$ may execute at this point with two options, writing '1' or '2'. Suppose that it writes '1', the I/O cache in fact does not write this message to the peer since the message is already in the cache. Instead, it only shades the associated data block to remember the state of the data stream (Figure 2f). The model checker continues running until the

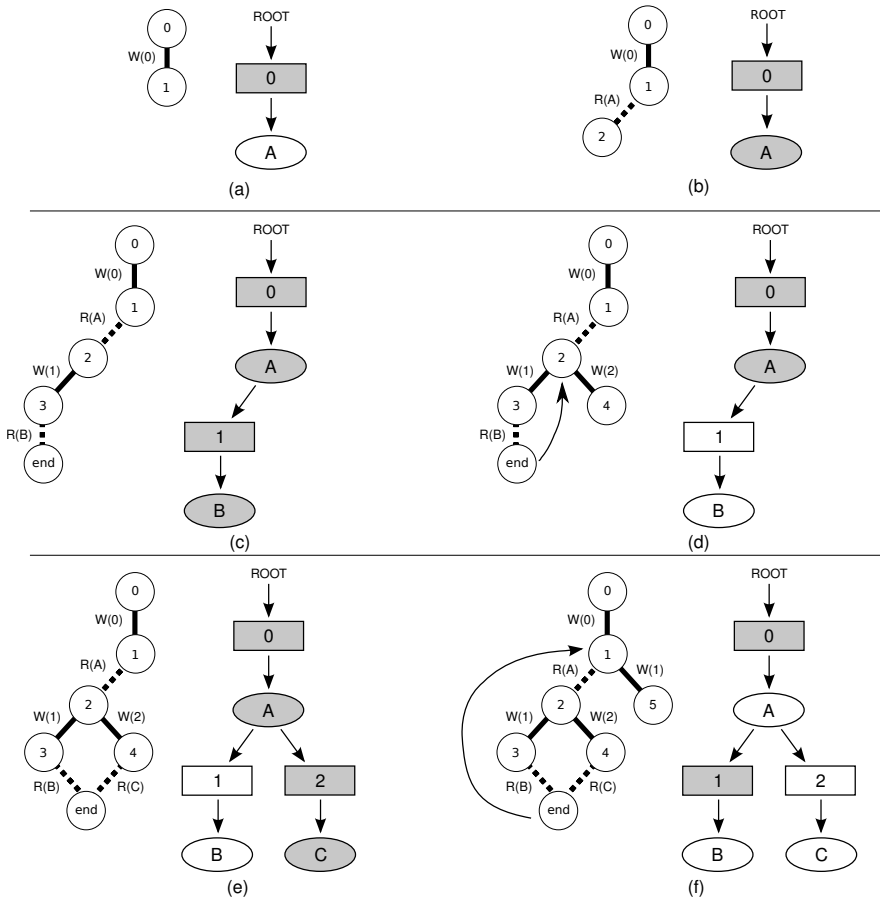



Figure 2: Evolution of the partial state space and cached traces. Two different communication traces are represented by solid lines and dashed lines. Rectangular nodes represent request messages. Circled nodes represent response messages.

whole state space is explored with the help of the I/O cache, which interacts with the SUT as a peer.

### B. Checkpointing Environments

Process checkpointing is a technique to create a *snapshot* of a group of processes. The snapshot stored in non-volatile memory is referred to as a *checkpoint*. A checkpoint can be loaded later on to recreate the process group in a certain state. After checkpointing, the recreated processes can continue running from where they were suspended as if they had not stopped running.

Most *virtualization tools* [17], [18], [19] provide checkpointing functions `save` and `restore`. However, virtualization consumes a large amount of system resources since it applies those functions on the entire system. Initially, we implemented our approach by using *Kernel-based Virtual Machine (KVM)* [18]. It took a few seconds merely to create one snapshot of a system, rendering it impractical.

A lightweight checkpointing package such as *MTCP* [20] can be used as a replacement in certain circumstances where a peer is a single-process program. The checkpointing package takes care of the state of a single process, unlike the virtualization tools. It approximately takes 200-300 milliseconds on average in order to create a checkpoint, which is acceptable. *Distributed MultiThreaded CheckPointing (DMTCP)* [21] is an extended version of MTCP, which manages a group of processes connected by network connections or parent-child relations. This work employs DMTCP as the checkpointing environment to support the I/O cache. All peer processes are controlled by DMTCP.

Checkpointing environments introduce a new method to synchronize a SUT and a peer. Model checkers save SUT states in order to backtrack it to any previously visited point in the state space. A checkpointing tool can do the same with peer processes. When the I/O cache has detected a new communication trace from the SUT, it restores the peer process from a checkpoint saved prior to the equivalent state instead of restarting the peer from the beginning. In the extreme case, we may create a peer checkpoint for each SUT state. In practice, the peer does not have to be checkpointed as often as the SUT. Some peer checkpoints may be omitted under a certain condition. An optimization method is discussed in Section III-D.

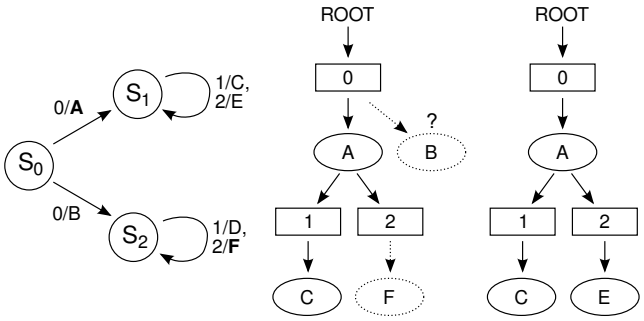



Figure 3: (Left) State transition diagram of a peer that produces non-deterministic output. (Middle) One possibility of incorrect cached data. (Right) Correct cached data.

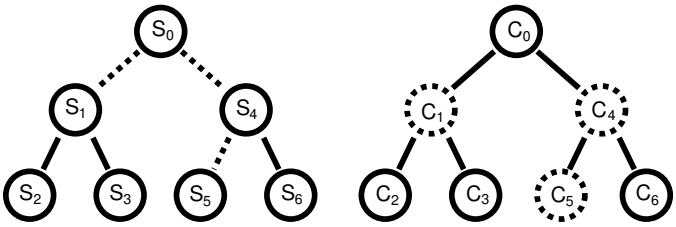


Figure 4: (Left) SUT state space. Dashed lines represent transitions without network I/O operations. (Right) Peer checkpoint space. Solid-line circles represent physical checkpoints. Dashed-line circles represent logical checkpoints.

### C. Support for Non-deterministic Peer Output

In this paper, we propose an approach to cope with non-deterministic peer output. An example of such a peer is shown in Figure 3 (left). The peer may change state and produce output differently ('A' and 'B') in each run, although it receives the same data '0' as shown in the transitions from $S_0$ to $S_1$ and $S_2$. The I/O cache approach without checkpointing restarts the peer process when the SUT produces a different trace after backtracking [12]. This technique does not work if the peer also produces non-deterministic output, which may cause inconsistency in the cached data. Suppose that the peer moves from $S_0$ to $S_1$, the I/O cache receives 'A' and 'C' from the peer. After backtracking the SUT produces a new trace ('02'), request message '2' is added into a new branch, forcing the peer to restart. The new peer may move from state $S_0$ to state $S_2$ after receiving message '0'. The transition to $S_2$ emits 'B', which differs from the existing cache content 'A'. The I/O cache may handle a mismatch by: (1) aborting the process, or (2) giving a warning and continuing. If it continues, it will receive 'F' as a response for '2'. The cache contents in Figure 3 (middle) indicate that the SUT receives response message 'AF' for request message '02', which is an incorrect behavior. According to the state transition diagram in Figure 3 (left), the peer obviously never produces 'A' and 'F' in the same run. The I/O cache may return an incorrect response message, because the new peer does not stay in the same state as its previous instance did. As a result, the model checker would report a false positive due to the communication trace that the SUT never receives in a normal environment.

This inconsistency can bring about a serious problem if the communication between two programs depends on the results of non-deterministic peer operations. For example, a SUT and a peer may perform *key exchange* [22] by generating random values required to build a shared secret key. If the I/O cache restarts the peer later, the peer will generate a new random value for building the key. The key obtained from the new random value is different from what the SUT is holding. As a result, both programs cannot decrypt messages from the opponent after the peer has restarted.

This issue can be solved by running a peer in a checkpointing environment, which can save and revert the peer state. A peer checkpoint is created as the SUT changes state. States $S_0$ and $S_1$ are saved for the example in Figure 3. When the SUT needs the peer in a certain state, the peer is restarted in a way that it will produce outcomes consistent with the cached contents. In this case, the peer is restarted from state $S_1$. The peer continues running from that point and correctly sends response message 'E' for request message '2'. The correct cache contents are shown in Figure 3 (right). By this method, only one behavior of the peer is revealed to the SUT as if the peer produced deterministic output.

### D. Checkpointing Strategies

Checkpointing every single peer state is not always necessary and not efficient since the I/O cache can replay cached messages in most cases. Instead, we introduce a concept of *logical checkpoints*, which do not occupy disk space. They are created as the model checker discovers new states of a SUT. Figure 4 shows *checkpoint space* as compared to SUT state space. State $S_i$ associates with logical checkpoint $C_i$.

A *checkpointing strategy* defines how to maintain the balance of the checkpoint creation overhead with the possibility of restoring a previous state directly. It decides whether to create a *physical checkpoint*, which occupies storage space, over the corresponding logical checkpoint. When the SUT needs the peer at a specific state, the model checker restores the corresponding logical checkpoint. If it lacks a physical checkpoint, the peer will be instead restored from the most-recent physical checkpoint on a path to that logical checkpoint. After that the model checker must replay communication data from there, up to the designated logical checkpoint.

Generally, creating two identical checkpoints is pointless. We assume that a peer does not change state significantly if it performs no network I/O operation, e.g. `connect`, `accept`, `send`, and `recv`. Following this assumption, the peer should be checkpointed only after a network I/O operation is performed. Using this strategy, an example of the resulting checkpoint space is shown in Figure 4. States $S_1$, $S_4$, and $S_5$ come from transitions without network I/O operations, so physical checkpoints are not created at $C_1$, $C_4$, and $C_5$. If the SUT needs the peer at $C_5$, we must start from physical checkpoint $C_0$ and replay network I/O operations, by using the cache contents, until it reaches $C_5$. A variant of this strategy is to only checkpoint after operation `connect` or `accept`.

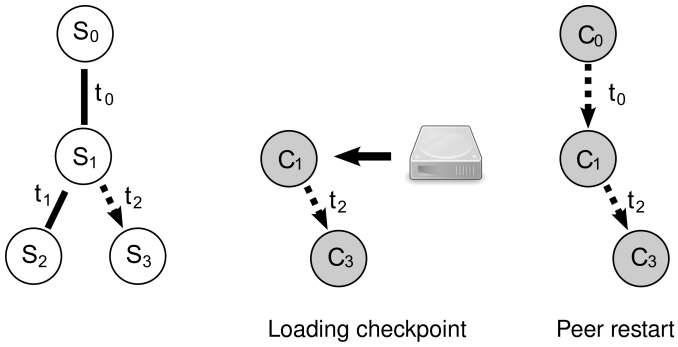



Figure 5: Two options : loading a checkpoint or starting a new peer.

In this case, the I/O cache must replay I/O operations from the beginning of the connection up to point where the peer is synchronized with the SUT again.

A checkpointing strategy takes effect after each SUT state transition to decide whether the current peer state should be saved. In addition to that, if the I/O cache receives a notification from the checkpointing environment about a non-deterministic operation, it will always saves the state of the peer. The I/O cache must do this in order to preserve the result of the non-deterministic operation. This method requires a way to detect non-deterministic peer "actions" at runtime. Our solution is to build wrapper functions for standard functions that may cause non-determinism such as `time` and `read`; see Section IV. When one of these functions is called with a certain argument, the wrapper function sends a notification to the I/O cache. Receiving the notification, the I/O cache saves the peer state after the current SUT transition is completed. Note that we must wait until the transition is completed in order to create a checkpoint synchronized with the SUT state as shown in Figure 4.

When a SUT needs a peer in one of the previous states, the I/O cache may either restore a peer from a checkpoint or start a new peer from the beginning. Figure 5 compares these options. The SUT moves from state $S_1$ to $S_3$, producing a hitherto unseen request message. $C_i$ is the peer state associated with SUT state $S_i$. Loading a checkpoint takes time in creating a process and the execution of transition $t_2$. Restarting the peer takes time in creating a new process and the execution of transitions $t_0$ and $t_2$. Checkpointing strategies should provide a way to estimate and compare cost in each choice. In the current implementation, the model checker always restores the peer from a checkpoint, assuming that loading the program space from a checkpoint is faster. In this case, the initial peer state ($C_0$) must always have a physical checkpoint since it can be a starting point to go to any logical checkpoints. Implementation of other checkpointing strategies constitutes future work.

### E. Restrictions

A checkpointing tool cannot force a peer process to produce output in a specific non-deterministic branch. The I/O cache uses the checkpointing tool only to make sure that the SUT receives peer output from a certain branch. However, the peer output captured by the I/O cache may be different in each run. As a result, only part of the SUT state space of the SUT is checked. In Figure 3, once the peer moves to state $s_1$, the SUT will never receive message 'BD' or 'BF' during the verification, although these messages are possible in a real run.

The introduction of checkpointing technology intervenes in the execution of a peer process in the sense that the peer must run in a special environment. In contrast to the pure cache-based approach, the behavior of the peer process in the new environment may differ from the original behavior. This limitation also implies that one must have a permission to set up a checkpointing environment on the machine that runs the peer process.

## IV. Implementation Architecture

Java PathFinder (JPF) [5] is a model checker for programs written in Java. It is used as the model checker and the run-time environment for SUT in this work. The pure I/O cache approach without checkpointing functions was developed as an extension of JPF called *net-iocache* [11], [23] for verifying networked applications. This work introduces process-checkpointing support by applying the tool called DMTCP [21] to suppress non-deterministic behaviors of peer processes. DMTCP runs a group of connected *nodes*, i.e. peer processes, in a special environment where some standard functions are wrapped in order to gain information to create system checkpoints. DMTCP has the *DMTCP coordinator* process that manages the execution of all nodes and handles external commands. When the DMTCP coordinator receives a `checkpoint` command, it captures the state of each node in the group, including connection information, in checkpoint files. One checkpoint file represents the state of one node, so for each `checkpoint` command, the number of checkpoints created is equal to the number of nodes currently running. The checkpoint files contain sufficient information to restart the group of processes at a state where each process is communicating with one another. In order to make DMTCP work with the I/O cache, we modify some part of DMTCP and register callback functions to capture the events inside the peer process.

### A. Connection with the Model Checker

DMTCP is a checkpointing tool for a group of connected processes. Users can add a process into the group by starting it with command `dmtcp_checkpoint`. Another way to add a process into the process group is creating a new process using the `fork`-family functions. Every child created by a process in the group automatically becomes a member of the group.

DMTCP saves the entire state of the process group including connections among the internal processes when receiving the checkpoint command. Similarly, it restarts all processes in a group from a given checkpoint when receiving the restart command. The SUT state is controlled by JPF while the peer state is controlled by DMTCP as shown in Figure 6. Since the

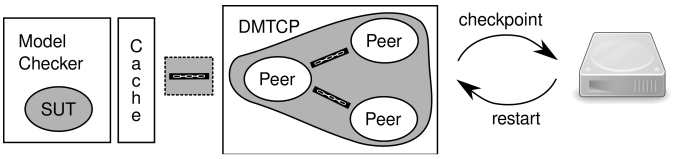



Figure 6: The state of a SUT and a group of peers is managed by the model checker and DMTCP, respectively, but the connection between them is not subject to checkpointing.

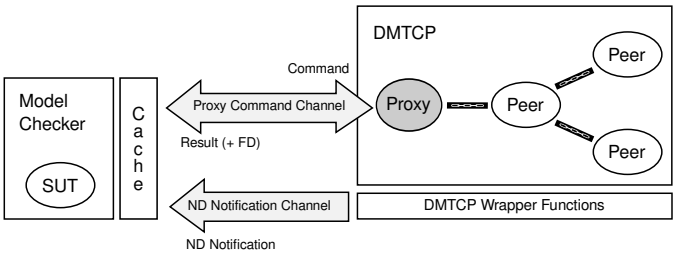



Figure 7: The proxy process represents the SUT inside the DMTCP environment. The I/O cache has two communication channels connected to DMTCP.

SUT is not a process in the group, the connection between the SUT and peers is not subject to checkpointing. As a result, the connection is closed when the I/O cache kills the group of peers before loading a new one from a checkpoint. When restarting, the I/O cache must provide a way to restore this connection so that the SUT and peer can communicate with each other again.

In our implementation, we create a *proxy* process that represents a SUT in the DMTCP environment. The proxy process runs similar to other peer processes as shown in Figure 7. When the SUT performs an operation that establishes a connection, the I/O cache sends the corresponding command to the proxy process. Currently, the proxy supports five commands: `create_socket`, `connect`, `accept`, `bind`, and `close`. Table I shows a mapping between the network operations called by SUT and the proxy commands. The proxy performs the requested operation and sends the result back to the I/O cache. Some operations may return a file descriptor that represents a network socket. The I/O cache can use the file descriptor it receives to communicate with the peer directly. In order to transfer file descriptors between processes, the SUT and proxy use a pair of Unix domain sockets to communicate with each other.

Table I: Supported Java methods and their associated proxy commands.

| Java Method | Proxy Command | FD Returned? |
|---|---|---|
| `new Socket()` | `create_socket` | yes |
| `Socket.connect()` | `connect` | no |
| `Socket.close()` | `close` | no |
| `new ServerSocket()` | `bind` | no |
| `ServerSocket.accept()` | `accept` | yes |

### B. DMTCP Modification

Our checkpointing-based approach requires a mechanism that notifies the I/O cache whenever a peer executes an instruction that causes non-deterministic behaviors. In order to implement such a mechanism, we need to watch calls to some functions of the peer program. In the current implementation, functions `time` and `read` are specially treated as they may produce non-deterministic results. Function `time` may be called obtain the current time, which varies across executions. This value is often used as a seed to generate a sequence of pseudo-random numbers such as function `srand`. When function `time` is called, the I/O cache is notified. As for function `read`, the I/O cache will be notified if the file descriptor argument is associated with the system random number generator device `/dev/random` or `/dev/urandom`. These devices are non-deterministic data sources supplied by the operating system.

DMTCP provides a set of wrapper functions that collects necessary information for checkpointing before calling the real version of the functions. The wrapped functions include both standard C libraries and system calls. In a similar way, we add one wrapper function (`time`) and modify an existing one (`read`). When either of these functions detects non-determinism (ND), it sends a *ND notification* to the I/O cache.

### C. Cache-DMTCP Private Communication

During verification, the I/O cache and DMTCP must have a way to communicate with each other. The I/O cache sends commands to the proxy process inside the DMTCP environment, as mentioned in Section IV-A. In addition to that, it must be ready to receive a notification when a peer process performs a non-deterministic operation.

We set up two communication channels between the I/O cache and DMTCP: the *proxy command channel* and the *non-determinism notification channel*, illustrated in Figure 7. When verification starts, the proxy process connects to the I/O cache using a Unix-domain socket, which allows the proxy to transfer file descriptors to the I/O cache. This connection is called the proxy command channel. It must be cut off before checkpointing, otherwise DMTCP will try to save the state of the process at the other side of the connection, i.e., the model checker. JPF does not run inside the DMTCP environment and should never be dumped into a checkpoint. The proxy command channel is re-established after checkpointing/restarting. We register the pre/post-checkpoint callback functions to DMTCP that are responsible for cutting off and repairing this connection, respectively.

The I/O cache recognizes non-deterministic operations on the peers by creating a worker thread that waits for ND notifications. The worker thread binds a TCP server socket to a fixed port number. Every time a peer executes a non-deterministic function, the corresponding DMTCP wrapper function asynchronously sends a ND notification packet to the worker thread as shown in Figure 7. If the I/O cache receives a notification during a transition, it will create a peer checkpoint at the end of the transition. Note that we must wait

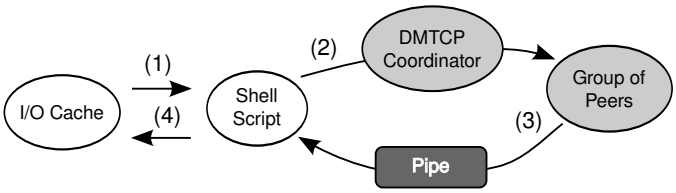



Figure 8: Communication between the model checker and DMTCP during checkpointing and restarting.

until the current transition is completed in order to generate the checkpoint state space that maps on the SUT state space one-to-one as shown in Figure 4.

When the I/O cache dispatches the checkpoint/restart command to DMTCP, it must be blocked until the peers are ready again. In the current implementation, the proxy process notifies the I/O cache via a named pipe (FIFO) as shown in Figure 8. The I/O cache executes a shell script (1) that dispatches a command to the DMTCP coordinator, an interface of the peer processes (2). After the operation has been completed, DMTCP notifies the I/O cache by putting a message in a pipe (3). Waiting on the pipe, the shell script receives the message and returns the control to the I/O cache (4). This procedure makes sure that the I/O cache only continues running when the peer side is ready. All processes must run on the same Linux machine in order to use the named pipe. Otherwise, another synchronization method must be provided. Currently, DMTCP supports only Linux-based operating systems, so our implementation adds no extra limitation.

## V. Experiments and Discussion

This section compares the time and the number of states generated in the model checking process between the pure I/O cache approach and the checkpointing approach with several checkpointing strategies. The experiment was run on an 8-core Mac Pro workstation with 24GB of physical memory, running Ubuntu 8.04, JPF 6 (changeset `382:4f9c3fc91a2f`), and DMTCP (revision 967). The time limit for each case was set to one hour. Table II shows the experimental results[2]. Column "no CP" denotes the I/O cache approach with no checkpointing support. Three checkpointing strategies were applied in the experiment.

1) **always save**: Create a checkpoint if the peer is alive.
2) **after I/O**: Create a checkpoint after a transition involved a networked I/O operation.
3) **after ND**: Create a checkpoint after a transition during which a ND notification is received.

In the alphabet application, a multi-threaded client sends number $n$ to the server and receives the $n$th letter of the English alphabet as a response, for a specified number of times. The alphabet client randomly sends a number of messages from set $\{0, 1, \ldots, 9\}$ while the alphabet server randomly sends either small or capital letters. Deterministic versions of the peers were used in the "no CP" case. Non-deterministic versions were used in the other cases. Note that the number of states explored by JPF is the same, regardless of determinism of peer output, since our approach captures one of the possible responses of the peer. The model checking process then runs as if the peer produced deterministic output.

The HTTP client simply requests a file from a server via HTTP. It generates worker threads to request multiple files in parallel. thttpd [24] is a small-size HTTP server, used in the experiment without modification. It sends static contents, thus deterministic output, according to client requests. Jget [25] creates multiple threads that each download a portion of a file in parallel from a server.

*ScpTo* is an example program in the *Java Secure Channel (JSch)* package [26], which copies a local file to a remote host via a secure channel. Both the client and server can produce non-deterministic output. ScpTo and the server generate a random value in the process of building a secret shared key [22]. As explained in Section III, checkpointing support is essential in this case. The GUI code in the program is removed before doing the experiment with Dropbear [27], a SSH server. The I/O cache with checkpointing has found a fault in ScpTo that involves a race condition. ScpTo creates a session thread to receive packets from the server while the main thread sends packets to the server. Both threads are not synchronized properly so that a race condition happens under a certain thread schedule. If the main thread makes progress much faster than another thread and reaches the point where a required packet has not been received, it throws an exception[3]. Another version of ScpTo is bug-fixed and further abstracted in order to finish the verification within reasonable time. An abstract SSH server runs as a peer for this version of ScpTo. Both versions were verified in the experiment together with other applications.

The performance of the checkpointing approach with the "ND" strategy is not much different from the pure cache approach (no CP) since it only creates a checkpoint if necessary. It also provides support for non-deterministic peers, making it more powerful than the previous version of the I/O cache. The "I/O" strategy is slightly slower than "ND", because it creates more checkpoints. However, it would be useful when the peer takes a long time in I/O operations since it prevents re-execution of those operations. The "always" strategy excessively creates checkpoints, so its performance is not practically useful. Its results are presented for the sake of comparison.

## VI. Conclusions and Future Work

Software model checkers cannot be applied directly to a program that interacts with external processes. Cache-based model checking allows a single-process model checker to verify such a program against an external process. This approach scales well, but imposes some requirements on the target system. In particular, previous work required peer processes to

[2]The verification time and the number of states are higher than the results in a previous publication [12] due to a change in JPF to cover more thread schedules.

[3]This bug has been acknowledged by the developer.

Table II: Experimental results

| SUT | Peer | #conn | #msg | time (mm:ss) | | | | #states | #checkpoints | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | always | I/O | ND | no CP | | always | I/O | ND |
| ND alphabet client | ND alphabet server | 2 | 2 | 27:45 | 0:11 | 0:10 | 0:05 | 7572 | 3438 | 6 | 4 |
| | | | 3 | > 1h | 0:26 | 0:24 | 0:14 | 33.3K | - | 12 | 7 |
| | | | 4 | - | 1:18 | 1:12 | 0:53 | 147.5K | - | 24 | 13 |
| | | 3 | 2 | - | 4:22 | 4:20 | 4:08 | 525.3K | - | 10 | 6 |
| | | | 3 | - | 33:40 | 33:22 | 32:38 | 4581.9K | - | 20 | 11 |
| | | | 4 | > 1h | | | | | | | |
| ND alphabet server | ND alphabet client | 2 | 2 | 2:14 | 0:11 | 0:10 | 0:01 | 299 | 277 | 9 | 8 |
| | | | 3 | 3:55 | 0:16 | 0:16 | 0:02 | 499 | 491 | 15 | 14 |
| | | | 4 | 5:53 | 0:21 | 0:20 | 0:03 | 747 | 739 | 20 | 19 |
| | | 3 | 2 | 33:59 | 0:18 | 0:18 | 0:04 | 4269 | 4241 | 15 | 13 |
| | | | 3 | > 1h | 0:28 | 0:27 | 0:07 | 8775 | - | 24 | 22 |
| | | | 4 | - | 0:37 | 0:37 | 0:10 | 15.5K | - | 32 | 30 |
| | | 4 | 2 | - | 0:48 | 0:47 | 0:29 | 57.8K | - | 21 | 18 |
| | | | 3 | - | 1:35 | 1:34 | 1:05 | 143.1K | - | 33 | 30 |
| | | | 4 | - | 2:50 | 2:48 | 2:09 | 295.2K | - | 44 | 41 |
| | | 5 | 2 | - | 7:06 | 7:06 | 6:34 | 746.4K | - | 27 | 23 |
| | | | 3 | - | 20:07 | 19:59 | 18:59 | 2209.1K | - | 42 | 38 |
| | | | 4 | - | 47:08 | 47:03 | 44:58 | 5295.8K | - | 56 | 52 |
| HTTP client | thttpd | 2 | N/A | 2:57 | 0:05 | 0:04 | 0:02 | 415 | 415 | 3 | 1 |
| | | 3 | | 46:42 | 0:51 | 0:50 | 0:48 | 6675 | 6675 | 4 | 1 |
| | | 4 | | > 1h | 23:12 | 23:07 | 22:10 | 112.5K | - | 5 | 1 |
| | | 5 | | > 1h | | | | | | | |
| Jget | | 2 | | 41:54 | 0:15 | 0:15 | 0:11 | 5984 | 5984 | 4 | 3 |
| | | 3 | | > 1h | 45:48 | 45:48 | 45:43 | 839.8K | - | 6 | 4 |
| ScpTo[1] | Dropbear | 1 | | 7:56 | 0.15 | 0:15 | ✘ | 1027 | 1026 | 9 | 9 |
| ScpTo (bug fixed) | Abstract SSH Server | 1 | | 1:15 | 0:05 | 0:05 | ✘ | 167 | 167 | 6 | 2 |
| | | 2 | | > 1h | 9:21 | 9:19 | ✘ | 557.7K | - | 8 | 3 |

✘: The I/O cache without checkpointing does not support these cases.
[1]A bug is found in this case.

be deterministic. In this work, the class of verifiable programs has been expanded to cover non-deterministic peers, which are controlled by process checkpointing. Our extension creates checkpoints of a peer program according to a checkpoint strategy. The experiment has shown that the overhead caused by the checkpointing tool is acceptable if an appropriate strategy is used.

Future work includes development and analysis of checkpointing strategies. The shell scripts that communicate with DMTCP will be replaced with a library in the I/O cache to eliminate the platform dependency. We also have a plan to run each peer process under a model checker. The model checker could be modified so that it controls low-level peer behaviors such as thread scheduling. Furthermore, the model checker may store peer states in memory rather than non-volatile storage, reducing the I/O operation overhead. Being able to control thread scheduling, we could analyze the peer behaviors and selectively perform the ones that would potentially reveal faults in the SUT. We will consider how the model checker engines communicate with each other as well.

*Acknowledgment*

This work was supported by *KAKENHI* (23300004 and 23240003).

## REFERENCES

[1] A. Tanenbaum, *Modern operating systems*. Prentice-Hall, 1992.
[2] G. J. Myers, *The art of software testing*. New York : Wiley, 1979.
[3] T. Ball, S. Burckhardt, K. E. Coons, M. Musuvathi, and S. Qadeer, "Preemption sealing for efficient concurrency testing," in *TACAS' 10*, 2010, pp. 420–434.
[4] E. Clarke, O. Grumberg, and D. Peled, *Model Checking*. MIT Press, 1999.
[5] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, "Model checking programs," *Automated Software Engineering Journal*, vol. 10, no. 2, pp. 203–232, 2003.
[6] P. Godefroid, *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*, J. van Leeuwen, J. Hartmanis, and G. Goos, Eds. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1996.
[7] S. Ghosh, *Distributed Systems: an Algorithmic Approach*. Boston: Twayne Publishers, 2006.
[8] S. D. Stoller and Y. A. Liu, "Transformations for model checking distributed Java programs," in *SPIN '01: Proceedings of the 8th international SPIN workshop on Model checking of software*. NY, USA: Springer-Verlag New York, Inc., 2001, pp. 192–199.
[9] C. Artho and P. Garoche, "Accurate centralization for applying model checking on networked applications," in *Automated Software Engineering Conf.*, Tokyo, Japan, 2006, pp. 177–188.
[10] E. D. Barlas and T. Bultan, "NetStub: A framework for verification of distributed Java applications," in *Automated Software Engineering Conf.*, Georgia, USA, 2007, pp. 24–33.
[11] C. Artho, W. Leungwattanakit, M. Hagiya, and Y. Tanabe, "Efficient model checking of networked applications," in *Proc. TOOLS EUROPE 2008*, ser. LNBIP, vol. 19. Zurich, Switzerland: Springer, 2008, pp. 22–40.
[12] C. Artho, W. Leungwattanakit, M. Hagiya, Y. Tanabe, and M. Yamamoto, "Cache-based model checking of networked applications: From linear to branching time," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 447–458.
[13] Y. Nakagawa, R. Potter, M. Yamamoto, M. Hagiya, and K. Kato, "Model checking of multi-process applications using SBUML and GDB," in *Workshop on Dependable Software: Tools and Methods*, Yokohama, Japan, 2005, pp. 215–220.

[14] J. Dike, *User Mode Linux*. Prentice Hall PTR, 2006.
[15] R. Stallman, R. Pesch, and S. Shebs, *Debugging with GDB : the GNU source-level debugger*, 9th ed. Boston, MA : Free Software Foundation, 2002.
[16] P. Godefroid, "Software model checking: The VeriSoft approach," *Form. Methods Syst. Des.*, vol. 26, no. 2, pp. 77–101, 2005.
[17] "OpenVZ documentation," http://wiki.openvz.org.
[18] Red Hat, Inc., "KVM," http://www.linux-kvm.org.
[19] Oracle, "Virtualbox," http://www.virtualbox.org/.
[20] M. Rieker and J. Ansel, "Transparent user-level checkpointing for the native POSIX thread library for Linux," in *In Proc. of PDPTA-06*, 2006, pp. 492–498.
[21] J. Ansel, K. Aryay, and G. Coopermany, "Dmtcp: Transparent checkpointing for cluster computations and the desktop," in *Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–12.
[22] W. Diffie and M. E. Hellman, "New directions in cryptography," *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, November 1976.
[23] W. Leungwattanakit, C. Artho, M. Hagiya, Y. Tanabe, and M. Yamamoto, "Verifying networked programs using a model checker extension," in *ICSE Companion proceedings*, Vancouver, Canada, 2009, pp. 409–410.
[24] ACME Laboratories, "thttpd - tiny/turbo/throttling HTTP server," http://www.acme.com/software/thttpd/.
[25] S. Paredes, "Jget," http://www.cec.uchile.cl/~sparedes/jget/.
[26] JCraft, Inc., "JSch - Java Secure Channel," http://www.jcraft.com/jsch/.
[27] M. Johnston, "Dropbear SSH server and client," http://matt.ucc.asn.au/dropbear/dropbear.html.