Introduction of Virtualization Technology to Multi-Process Model Checking

Watcharin Leungwattanakit*
University of Tokyo, Tokyo, Japan
watcharin@is.s.u-tokyo.ac.jp

Cyrille Artho
RCIS/AIST, Tokyo, Japan
c.artho@aist.go.jp

Masami Hagiya
University of Tokyo, Tokyo, Japan
hagiya@is.s.u-tokyo.ac.jp

Yoshinori Tanabe
University of Tokyo, Tokyo, Japan
y-tanabe@ci.i.u-tokyo.ac.jp

Mitsuharu Yamamoto
Chiba University, Chiba, Japan
mituharu@math.s.chiba-u.ac.jp

Abstract

Model checkers find failures in software by exploring every possible execution schedule. Java PathFinder (JPF), a Java model checker, has been extended recently to cover networked applications by caching data transferred in a communication channel. A target process is executed by JPF, whereas its peer process runs on a regular virtual machine outside. However, non-deterministic target programs may produce different output data in each schedule, causing the cache to restart the peer process to handle the different set of data. Virtualization tools could help us restore previous states of peers, eliminating peer restart. This paper proposes the application of virtualization technology to networked model checking, concentrating on JPF.

1 Introduction

Recently, software model checking [8] has become widespread. It is not only applied to application models but also actual implementations. However, many modern applications are very complex since they exchange data and interoperate with other entities via a network. Several techniques have been established to verify networked applications by a model checker. Centralization [4, 11] transforms every relevant process into a thread and verifies the application as a single process. However, this technique suffers from the state explosion problem, caused by the large number of interleavings among large numbers of processes and threads. NetStub [6] simplifies the complexity of the process by replacing peer processes with stub objects. The target process then talks with mock processes written only for verification. However, this approach still requires that a user write a stub for each peer process. Finally, Nakagawa et al. [10] proposed a technique that verifies every process in an application inside a multi-process model checker. This technique does not scale well because of the huge number of process interleavings, but the tool has shown that existing virtualization environments are feasible and efficient.

A single-process model checker like Java PathFinder (JPF) [9], without modification, cannot be applied to networked applications since some related processes are running outside the model checker and not backtracked together with the target process. For instance, two threads, T1 and T2, execute code shown in Figure 1. They write a character to and read a character from a peer process. The other side of the figure shows a subset of the state space generated by a model checker. Initially, the model checker executes the entire program under the first schedule (A). However, when it backtracks the program to state 1 and executes another branch, T2 sends a duplicate character to the peer process in the transition to state 4 (B). The duplicate characters may interfere with the correct functionality of the peer process, which is not backtracked. Furthermore, when T1 tries to read a character from the peer again in the transition 4–5 (C), the peer may reply nothing to the target process. This causes T1, and consequently the model checker, to wait forever for input without producing a verification result.

E. Denney, D. Giannakopoulou, C.S. Păsăreanu (eds.); The First NASA Formal Methods Symposium, pp. 106-110
*This work was supported by a *kakenhi* grant (2030006).

T1	T2

void run() {
W1: write("1");
R1: read();
}
	void run() {
	W2: write("2");
	R2: read();
	}

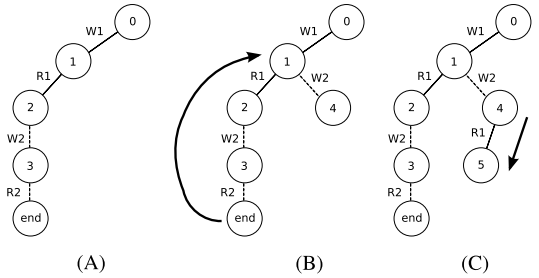

Figure 1: Left: Example networked program. Right: Development of the partial state space.

Our previous work [5] solves this problem by using a cache to capture the data transferred between a program under test and its peer process. Only one process is executed inside the model checker, whereas the other process runs on the host machine in native mode. The cache replays captured data appropriately so that the target program is given all necessary responses. Peers do not realize that they are connected to a process that runs in a model checking environment. The basic concept of the cache works well for simple request-response networked programs, but non-trivial programs need special care.

Some networked programs respond differently each time they receive a request. For example, dynamic web pages are reprocessed every time they are requested; the reproduced pages may differ from each other. Such non-determinism makes it impossible to replay cached data. Our current implementation solves this problem by having the cache create a new instance of the peer process to accept a new communication trace. Instead of restarting, it would be possible to run the peer process inside a virtual environment that the program state can be saved and restored. This method could reduce the time spent in starting a new process and control the number of connections generated by the cache. In this paper, we ignore non-determinism in peers for scalability. The cache approach compromises on soundness by verifying the target process with a subset of possible responses from peers, since we cannot determine the degree of non-determinism of a process running outside the model checker.

2 Concept

2.1 Virtualization

Virtualization is a technique that emulates several computing platforms on one physical machine. The virtualized systems use the computing resources of operating system (OS) on the host machine. There are several types of virtualization: *hardware emulation*, *full virtualization*, *paravirtualization* and *OS-level virtualization*. Hardware emulation creates virtual hardware devices required to run an unmodified guest OS. Full virtualization uses software called *hypervisor* as a mediator between guest systems and physical hardware. Guest systems may access physical hardware directly except for certain protected instructions, which are handled by the hypervisor. Paravirtualization offers better performance [7] by hosting modified guest systems, which are aware of the virtualization environment. OS-level virtualization requires a specially modified kernel to manage guest systems. Every guest is executed on the host OS, yielding nearly the same performance as in native mode. A general architecture of a virtualized system is shown in Figure 2. A *node*[1] represents a physical machine running both host and guest operating systems. The *hypervisor* is the layer of software that manages every guest system to run simultaneously. Each guest operating system is contained in a *domain*, a space that the guest system is allowed to access.

[1] These terms may be used differently in the documentation of some virtualization tools.

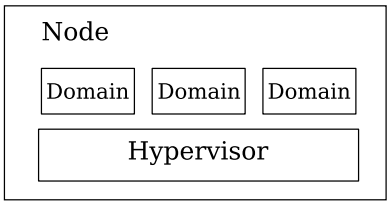

Figure 2: Architecture of a virtualized system.

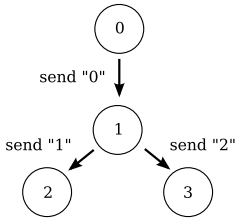

Figure 3: Part of the search space of the non-deterministic alphabet client.

2.2 Peer Processes in Virtualized Environment

Non-deterministic processes may send different data streams for each thread schedule. An example client/server system implemented as an alphabet client/server illustrates this. After connecting to the server, the client writes characters to its output data stream and reads characters from its input data stream. Read and write instructions are called separately by two threads, which are interleaved arbitrarily. The writer thread sends messages in set $N = \{\text{“0”}, \text{“1”}, \ldots, \text{“9”}\}$ to the server. The server side, when receiving a message $m \in N$, replies $f(m)$ to the client where function $f = \{(\text{“0”}, \text{“}a\text{”}), (\text{“1”}, \text{“}b\text{”}), \ldots, (\text{“9”}, \text{“}j\text{”})\}$. If the client sends “0” in the first step and randomly sends “1” or “2” in the next step, the search space that the model checker generates is as shown in Figure 3. After the model checker has finished the left branch, we may create a new connection to the server and re-send “0” followed by “2” to handle the other possibility. However, if the server is run inside a virtualization environment, its state could be stored and re-used later for backtracking.

The common operations of virtualization include `suspend`, `resume`, `kill`, `dump` and `undump`. Operation `suspend` temporarily pauses execution of a guest system, and `resume` starts execution again. `Kill` forcibly shuts down a guest system. `Dump` saves the current state of the running machine as a file, whereas `undump` recovers a program state from a file. Virtualization comes into play when the model checker encounters a decision in the search space. A model checker extension may call `suspend` and `dump` to keep peer's state before the decision. When backtracking, the extension should “kill” the current peer process and call `resume` and `undump` to restore the peer to the state before the decision.

Care should be taken when saving a peer state so that the overhead from dumping the state does not dominate the time saved by preventing peer restarts. A hybrid solution combining caching and virtualization on the same system, may constitute the ideal trade-off. One could dump only some of the states, for example, the states after an I/O operation and/or an expensive operation. As long as the model checker does not backtrack to a state prior to the saved program state, these steps need not be repeated. The interaction with the peer will be only replayed from the last saved state, instead of the beginning, to the present state. Communication data needed for replaying execution can be stored in the cache.

2.3 System Architecture

The virtual machine hosting a domain needs to be configured beforehand. A virtual machine controller, which is a part of our extension, will handle this task. The peer including its configuration must be prepared to run inside a guest system. Users would write a shell script specifying how to start and configure a peer.

Peer processes may run either on a remote machine or the same machine as the target program. In normal testing, the target program connects directly to every peer process on the local host and the remote host. Each peer can access its *environment,* such as the file system and system variables of the host, that it runs on (see Figure 4, left). If the target program is verified by JPF, each peer will be run

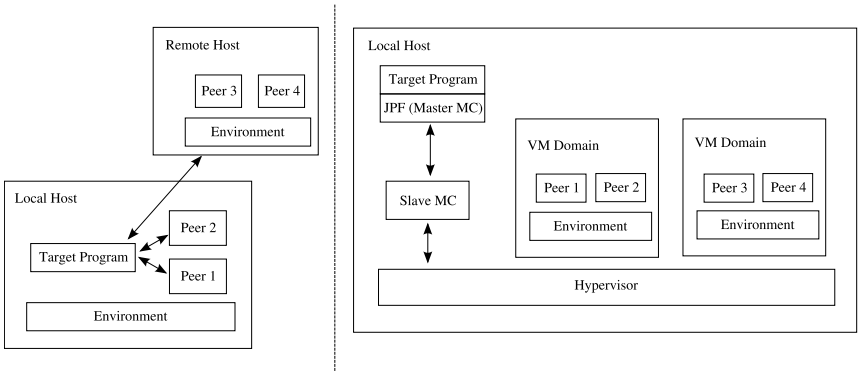

Figure 4: Left: Normal testing setup. Right: Model checking setup.

in the domain representing the host that it originally ran on (see Figure 4, right). Peers 1 and 2 are run in a domain representing the local host, whereas the peers on the remote host, peers 3 and 4, are run in another domain. Both domains, which contain the environment of the peers, are controlled by the same hypervisor. JPF, the *master model checker*, interacts with the *slave model checker*, which acts as an interface to peers. When a certain condition is satisfied, the slave model checker will save the state of every peer via the hypervisor. The slave model checker also restores previous states when JPF backtracks and replays I/O operations if necessary. This method differs from the multi-process model checker approach [10], in which the model checker checks every interleaving among all processes. In our proposed approach, JPF only explores thread interleavings in the target process, whereas the slave model checker synchronizes peer's states without investigating any interleavings.

2.4 Benefits of Virtualization

Creating extra connections would be a considerable overhead for programs with a high degree of non-determinism. Virtualization could make this process simpler by executing each peer process inside a virtual environment, where it is possible to save and restore the state of the peer process. For example, in Figure 3, the cache saves the state of the server at node 1 before exploring the left branch. This saved state is restored when the model checker backtracks to node 1, where it is about to traverse the right branch. The cache does not need to create extra connections during search thanks to virtualization.

Performance in model checking would be improved when using a peer with high startup or processing time. Some networked programs take a considerable amount of time to load necessary modules and configuration files before being ready to handle requests. If such programs are used as a peer process without the help of a virtualization tool, the model checker will waste a considerable amount of time on restarting the peer.

Furthermore, the state of the peer becomes open to inspection when the peer runs inside a virtualized environment. In the current implementation, the cache waits a certain amount of time for a peer response after it has sent a message [5]. If no response has arrived, it assumes that the message sent so far is not a complete request that elicits a peer response. However, this assumption is not always true. Responses may come late due to slow request processing, or network congestion in case of the peer running remotely. It is difficult, if not impossible, to determine the status of a peer running in an environment that we do not control. Future work includes development of a method to determine whether a peer has finished producing output.

3 Conclusion and Ongoing Work

This paper shows the application of virtualization technology to multi-process model checking. The idea could be applied on any model checking system that executes actual processes during verification. Peer processes are executed under virtual environments rather than the host operating system. When a model checker backtracks the verified process, creation of extra connections is not necessary. Instead, the program state is recorded occasionally and is restored when backtracking. This idea will be implemented in a JPF extension called *net-iocache*.

This extension for JPF 4 has been developed to verify networked applications without virtualization, which is ongoing work [5]. We are now studying the strengths and weaknesses of each virtualization tool. Important factors like the speed of saving/restoring system states and size of a saved state will be tested and measured before selecting a tool. It is also preferable to store program states in memory rather than a disk to minimize overhead. Unfortunately, we have not found such a function in any tools that we are studying. We may consider adding this feature if necessary. Our development platform is Ubuntu Linux 8.04, for which the *Linux Kernel Virtual Machine* (KVM) [1] is one of the candidates of virtualization tools. It allows guest operating systems to run in the user-space of the host kernel. Although it requires a certain feature of hardware, KVM gives us a performance of guest programs close to running natively on the host system. Other candidates include Xen [3] and OpenVZ [2].

References

[1] KVM documentation. `http://kvm.qumranet.com/kvmwiki`.

[2] OpenVZ documentation. `http://wiki.openvz.org`.

[3] Xen web page. `http://www.xen.org`.

[4] C. Artho and P. Garoche. Accurate centralization for applying model checking on networked applications. In *Automated Software Engineering Conf.*, Tokyo, Japan, 2006.

[5] C. Artho, W. Leungwattanakit, M. Hagiya, and Y. Tanabe. Efficient model checking of networked applications. In *Proc. TOOLS EUROPE 2008*, volume 19 of *LNBIP*, pages 22–40, Zurich, Switzerland, 2008. Springer.

[6] Elliot D. Barlas and Tevfik Bultan. NetStub: A framework for verification of distributed Java applications. In *Automated Software Engineering Conf.*, Atlanta, Georgia, USA, 2007.

[7] V. Chaudhary, Minsuk Cha, J.P. Walters, S. Guercio, and S. Gallo. A comparison of virtualization technologies for HPC. *22nd International Conference on Advanced Information Networking and Applications*, pages 861–868, March 2008.

[8] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[9] Klaus Havelund and Thomas Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[10] 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*, pages 215–220, Yokohama, Japan, 2005.

[11] Scott D. Stoller and Yanhong A. Liu. Transformations for model checking distributed Java programs. In *SPIN '01: Proceedings of the 8th international SPIN workshop on Model checking of software*, pages 192–199, New York, NY, USA, 2001. Springer-Verlag New York, Inc.