Proceedings of the 2019 Winter Simulation Conference
N. Mustafee, K.-H.G. Bae, S. Lazarova-Molnar, M. Rabe, C. Szabo, P. Haas, and Y.-J. Son, eds.

USING SIMULATION TO QUANTIFY THE RELIABILITY OF CONTROL SOFTWARE

James Nutaro
Ozgur Ozmen

Computational Sciences and Engineering Division
Oak Ridge National Laboratory
1 Bethel Valley Road, MS 6085, P.O. Box 2008
Oak Ridge, TN 37831, USA

ABSTRACT

More than two decades ago, Butler and Finelli examined the problem of experimentally demonstrating the reliability of safety critical software and concluded that it was impractical. We revisit this conclusion in the light of recent advances in computer system virtualization technology and the capability to link virtualization tools to simulation models of physical environments. A specific demonstration of testing for reliability is offered using software that is part of a building control system. Extrapolating the results of this demonstration, we conclude that experimental demonstrations of high reliability may now be feasible for some applications.

1 INTRODUCTION

Butler and Finelli (1993) examined the problem of testing software for reliability, and they concluded that the cost and time required to execute sufficient numbers of tests precluded practical demonstrations of mean time to fail. To illustrate the problem, Butler and Finelli offered a simple model of the time required to demonstrate a mean time to failure μ. The testing protocol involves creating n replicates of the system to be tested and allowing these to run until $r \leq n$ are observed to fail. When a failure occurs, a new test is started so that there are n active tests at any time. Having observed r failures over a total time of operation t, the mean time to fail is estimated by t/r. Increasing r and n improves the statistical significance of this estimate. The total testing time D needed to observe these r failures is

$$D = \mu \frac{r}{n} \ . \tag{1}$$

This estimate relies on several simplifying assumptions concerning the statistical process of failures. For example, failures must be uniformly distributed over the space of possible software inputs and the consequent binomial failure process in time is adequately approximated by an exponential distribution. It also overlooks the difficult problem of test case design. Regardless of these issues, Equation (1) illuminates a primary obstacle to demonstrate large μ, which is the need for a very large n to make D practical.

Replicating software does not pose a serious challenge to achieving large n. Rather, the challenge is to have many instances of the environment where the software works. We may have as many instances of a simulation model of the environment as needed and so having such a model overcomes the challenge in principle. However, the simulation is hobbled by either: (i) the need to run quickly enough to simulate software running on actual hardware, or (ii) interactions with a model of the software, or some subset of the software that can be incorporated into a model, rather than the software as it will be put into operation.

These two problems can be solved by executing the software as it will be put into operation on a model of its computational hardware. Because the simulated hardware can become a part of the simulation

model of the environment, n is limited only by the availability of computational resources for running simulations. This idea was proposed by Hu and Zeigler (2005) and given the name *model continuity*. They developed a technical approach by which software is developed as part of a simulation model and then seamlessly transitioned into the operational system. A constraint of this solution is the specialized software development environment that supports the transition.

Over the last decade there has been considerable progress made towards simulating computer hardware with sufficient speed and fidelity to execute complete software stacks - comprising operating system, device drivers, and applications - within a larger, simulated environment. In principle, this can eliminate the need for specialized software development tools while still enabling the model continuity concept.

Recent work towards building these types of simulations has taken two approaches. One is to modify a hypervisor, turning it into a simulation engine (Lee et al. 2014; Yoginath et al. 2015; Gupta et al. 2011). The other is to use a computer system emulator that has simulated hardware advancing in step with an overarching simulation clock (Chiang et al. 2011; Montòn et al. 2007; Montòn et al. 2009; Kurimoto et al. 2013; Mueller and Pétrot 2011; Weingärtner et al. 2008). The purpose of these prior efforts was to avoid modeling elements of a software system. For instance, the ability to insert fielded software into a simulation is essential for simulating attacks on computer systems and for writing software that uses hardware which is not yet available (Pantoquilho 2010; Shankar et al. 2014).

Here we examine how this type of technology could be used to test for software reliability. For this purpose, we have modified the QEMU (Quick Emulator) computer system emulator (QEMU 2019) for use as a component in a general purpose discrete event simulation, and then we present a case study in which this emulator is used to exercise a control system. By extrapolating the results of this exercise, we look at the time and computational resources that would be necessary to experimentally demonstrate a specific mean time to fail.

2 THE QEMU COMPUTER SYSTEM EMULATOR

QEMU is a computer system emulator for executing real software on simulated hardware. The simulated hardware includes memory, system buses, timers and clocks, and peripheral equipment such as disks, network cards, video cards, mice, keyboards, and serial ports. The simulation time for these subsystems is managed with a future event list called the virtual timer, and it is used just like the event queue in a discrete event simulation. When the time of the next future event is reached, QEMU stops execution of the emulated computer and invokes an event handler. Execution of the emulated computer resumes when the event handler returns.

Crucial for our purposes is that: (i) event handlers manage the states of simulated timing hardware, and (ii) the emulated microprocessor is halted while an event handler is running. Following an approach similar to the one used in QBox (Mueller and Pétrot 2011) to synchronize the virtual timer within QEMU with the simulation clock of a SystemC simulation tool (IEEE Standards Association 2012), we exploit the virtual timer to use QEMU as a component in a discrete event simulation.

The emulated microprocessor in QEMU can execute instructions in one of two ways. In the first, a sequence of machine instruction sequence is broken into chunks called translation code blocks. These are passed to a just in time compiler that translates the code blocks into instruction sequences for the computer on which QEMU is executing. In the second, QEMU passes instructions to the Linux Kernel Virtual Machine (KVM) for direct execution on the available hardware.

The just-in-time compiler is enabled when QEMU operates in its *icount* mode. In this mode, time within the emulator is advanced by a fixed (usually one) number of nanoseconds for every machine instruction that is executed. When an event handler is invoked, the emulated microprocessor stops immediately and, because time is advanced by the execution of instructions, the instruction execution rate does not deviate from the desired rate.

The *icount* mode can be configured such that when the emulated computer has no work to do, the simulation proceeds forward in time to the next event in the virtual timer's event list. Hence, when the

emulated hardware is idle, QEMU proceeds as a discrete event simulation by making instantaneous jumps through virtual time. This mode is particularly interesting because it allows software that is frequently idle to believe it is running in real time even though the simulation clock advances faster than real time.

On the other hand, the execution of a each instruction by the emulated microprocessor requires numerous instructions for the physical processor on which the just in time compiler executes. Hence, computationally active instruction sequences will cause the simulation to run much slower than real time.

The KVM overcomes this problem by using the physical microprocessor as the emulated microprocessor, thereby skipping the translation step and achieving near real time execution speeds. In this mode, time advances at the wall clock rate while the emulated microprocessor is processing instructions. To synchronize time with a discrete event model, we halt the emulated microprocessor while the synchronization protocol is executing. This is done by asking the operating system to suspend threads that are processing guest instructions.

The delay between this request and the operating system acting upon it can cause the emulated microprocessor to overrun the time of next event. So while this mode of execution offers near real time performance, it admits the possibility of intermittent error in the relationship between simulation time and time as perceived by software running on the emulated microprocessor.

QEMU was designed as a stand alone virtualization tool, without need for a capability to coordinate its internal simulation clock (called a virtual clock by the QEMU documentation) with some other simulation system. Therefore, it was necessary to retrofit the QEMU software with this capability (a patch to add this new feature in QEMU 2.11 is available at https://web.ornl.gov/~nutarojj/adevs/qemu-2.11.0.patched.zip). The protocol for coordinating advancement of time within QEMU and another discrete event simulator is illustrated by the sequence diagram in Figure 1. On start up, the modified QEMU software attaches to a UNIX domain socket, which we will call the synchronization socket, that has been created by the discrete event simulator. QEMU waits for a time advance value to be written to this socket by a model within the discrete event simulation that is QEMU's proxy. Upon receiving the time advance, QEMU schedules a virtual timer event for that interval into the future. The emulator executes instructions until that virtual timer event occurs.

When the event happens, QEMU halts the emulated microprocessor, records the simulation time that has actually elapsed when execution halts, and writes that time to the socket. In the *icount* mode, this time will match the scheduled event time; with KVM, the elapsed time may exceed the scheduled event time. In the latter case, the error is corrected by scheduling a catch up event in the discrete event simulator and then allowing QEMU to execute again after the catch up event has occurred. This process repeats until the simulation is terminated.

The software executing on the emulated computer can interact with a simulated environment via an emulated serial port and network interface card. Communication between these QEMU device models and the discrete event simulator also occurs through a UNIX domain socket. During the execution of a time advance by QEMU, the simulated devices may write data to this socket. In the case of the network card, these data are Ethernet frames sent by the software executing on the emulated computer. In the case of the serial port, these data are sequences of characters written to the serial device by the software running on the emulated computer. When QEMU completes its time advance, the discrete event simulator reads these data from the appropriate UNIX domain sockets and it writes to these sockets any frames or characters that have been transmitted to the emulated computer.

If large amounts of data are to be exchanged between QEMU and the discrete event simulator, it is possible that the buffers backing the UNIX domain sockets will become full and cause the sender to block. This will cause the simulation time to stop advancing if the intended recipient is waiting for a time advance to complete before reading from the socket. This problem is eliminated by having the discrete event simulator dedicate a thread to reading and writing each socket.

The reading thread loops on a blocking read of the socket. As messages become available, these are placed into a queue from which the discrete event simulator will extract them at the next synchronization

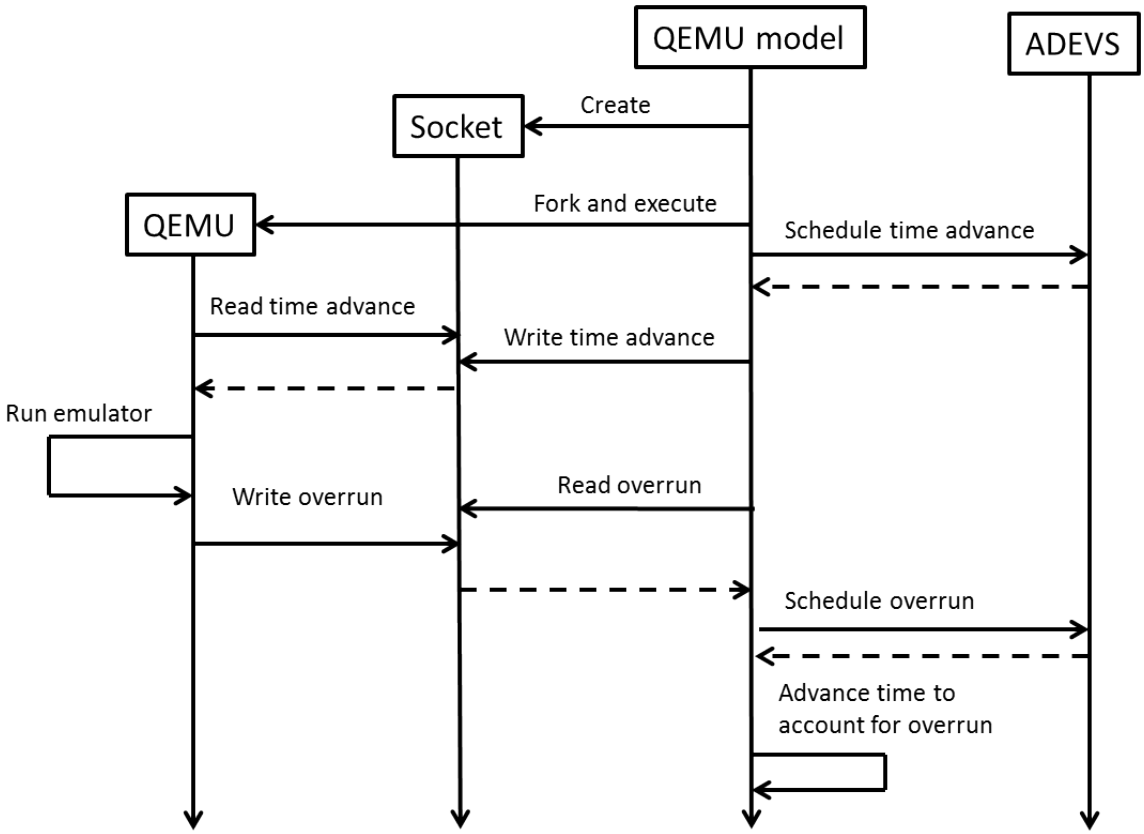


Figure 1: Time synchronization between QEMU and the overarching simulator.

point. The writing thread accepts data to be sent to the QEMU device and queues this for transmission. It writes queued messages to the socket using a blocking write. In this way there is always a thread available to extract data from the buffer and to write data to the buffer while the simulator advances.

When using this synchronization protocol, timing errors can result from two sources. One source of error is the timer overrun that can occur when using KVM. The other source of error is a delay of messages exchanged between the emulated computer and discrete event simulator. With the proposed scheme, data is exchanged at the points of synchronization between QEMU and the simulator. A necessary consequence of this approach is that messages will experience a delay between zero and single time advance plus the overrun. This error could be further exacerbated by delays imposed by the UNIX domain sockets and the scheduling of threads that monitor those sockets. If these delays are longer than the real time required to execute a time advance, then the difference will cause an undesired delay in the data exchange equal to the number of time advances that elapse while communication operations are completed.

3 TEST ENVIRONMENT FOR A BUILDING CONTROL APPLICATION

The software under test is a control system that coordinates the operation of air conditioning (HVAC) units within a building to maintain the temperature of each air conditioned zone near its set point while satisfying a limit on the number of units operating simultaneously. The effect of the control is to reduce peak electrical load (Nutaro et al. 2016). The control system is realized by a centralized software system. The centralized control software communicates with thermostats in the building via a serial line to read the temperature set point and temperature sensor and to turn the air conditioning unit on or off.

For the simulation based test, a building model was constructed based on the four zone building in which we have deployed the control system. This building and its model are described in (Ozmen et al. 2017). The building model was developed in Modelica and exported as a Functional Mock-up Unit (FMU). This mathematical model relates the zone temperatures to time varying outdoor air temperature and solar irradiance incident on the building, insulation and heat capacity of the building structure, and heat removed by the air conditioning units when they are cooling. The MODBUS RTU232 protocol is used as a serial communication interface between the control application and physical environment. Therefore, the building model includes an emulation of the serial port communication protocol by which the control software and real thermostats interact. Our modified QEMU contains an image of the control system software and its operating system.

Figure 2 shows the components of the simulated test environment. A Discrete EVent system Simulator (ADEVS) is used as the overarching simulation environment (Nutaro 2011). The modified QEMU is a component model within ADEVS, as are models of the MODBUS enabled thermostats and the FMU that implements the physical dynamics of the building. The FMU equations are solved using numerical algorithms for hybrid systems that are part of the ADEVS simulation package.

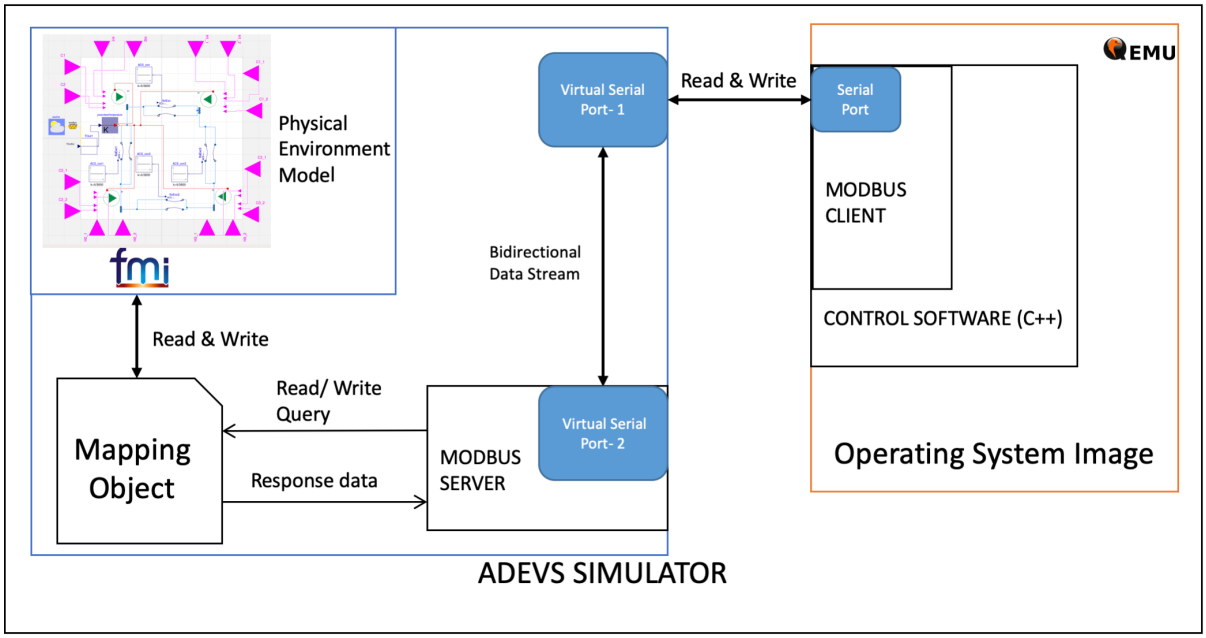


Figure 2: Components of the simulated test environment.

3.1 TEST PROTOCOL AND RESULTS

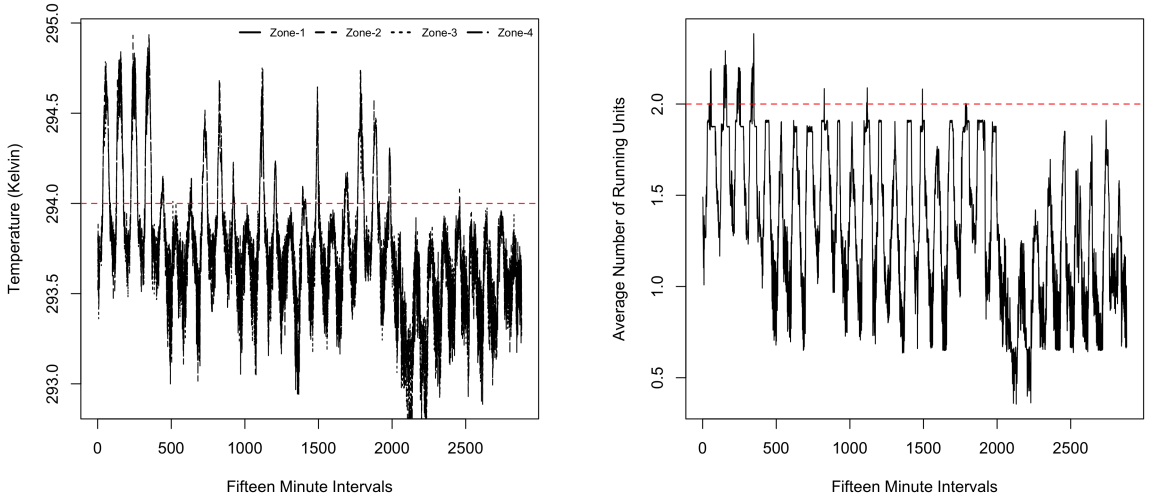
The test has two purposes. The first is to quantify the frequency and extent to which the control software cannot meet its goals of limiting the number of simultaneously operating units while keeping the temperature near the set point. To this end, we track two metrics: (i) the proportion of time any zone stayed one degree Kelvin above the temperature set point (294 K, that is approximately 69.5 Fahrenheit); and (ii) the proportion of time that more than two units run simultaneously.

The second purpose of the test is to demonstrate that the modified QEMU interacting with a model building can exercise the control system in a faster than real time simulation. This is relevant to our specific control system because it is prohibitively expensive to spend a week or more observing the system in the field to understand the full impact of a software change. More generally, the combined capability to

replicate the test and to execute it faster than real time is what can enable testing for high reliability, which we consider later in Section 4.

Historical weather data for Chicago, Illinois is used in the simulated environment (O'Hare Airport, 2013). Three tests are run, with each set covering part of a 30 day segment in the summer season (July 15 - August 15). The first test operates the system over the first ten days, the second test operates the system over 20 days, and the third test over the full 30 days. Because the tests are performed in a simulated summer, only cooling performance is considered. HVAC relay settings and the temperature readings are recorded at one second intervals and the collected data is aggregated to the aforementioned metrics over a test period.

Figure 3(a) presents the mean temperatures of the zones around the set temperature (red dashed line), and Figure 3(b) presents the mean number of units running around the maximum number of units the control software aims to limit (red dashed line). One processor and one gigabyte of memory is provided for the emulated computer, the time advance for the emulator is $1/10$ seconds, and the processor is simulated using the icount mode.



(a) Temperature for different zones over time.

(b) The number of simultaneously running units over time.

Figure 3: Simulation-based testing results.

Table 1 shows statistics from each test for the proportion of time any zone was one degree Kelvin above the set point temperature during the test. The confidence interval for the performance metric naturally shrinks with an increasing number of test days. Similarly, Table 2 shows shrinking confidence intervals for our estimate of the proportion of time the number of simultaneously running units is above two.

Table 1: Proportion of testing time that software has not fully satisfied the comfort (temperature) needs.

Testing Period	**Mean**	**Std. Deviation**	**95% Confidence Interval**
10 days	0.0836	0.4877	0.0010
20 days	0.0555	0.4834	0.0007
30 days	0.0372	0.4731	0.0006

Table 2: Proportion of testing time that peak load was greater than 2.

Testing Period	Mean	Std. Deviation	95% Confidence Interval
10 days	0.0130	0.6922	0.0015
20 days	0.0071	0.7190	0.0011
30 days	0.0047	0.7771	0.0009

What is most significant about these tests is the *real time* needed to obtain the given periods of simulated operation. The 30 day test required only 1.25 days of actual execution time, which is approximately 24 times faster than real time.

4 IMPLICATIONS FOR SAFETY CRITICAL SOFTWARE

For safety critical software, a mission duration m is typically between one and ten hours (Butler and Finelli 1993). For these types of systems, Butler and Finelli used Equation (1) to estimate the required hours of experimental operation that would be needed to demonstrate a given failure rate. A requirement for ultra-reliability would make the probability p of system failure during a mission on the order of 10^{-7} to 10^{-9}. The Butler and Finelli estimates assume an exponentially distributed time to obtain

$$\mu = -\frac{m}{\ln(1-p)} \ .$$

Suppose we wish to test for no more than one failure per μ hours of operation and we want to observe r such failures. If each of n replicated simulations runs s times faster than real time, then the number of real hours needed to demonstrate the desired failure rate is

$$D = \mu \frac{r}{sn} \text{ hours.} \tag{2}$$

Assuming a ten hour mission time and $p = 10^{-9}$, we have $\mu \approx 10^{10}$ hours, which is $1,141,550$ years. Estimates using Equation (2) for several p assuming $m = 10$, $s \approx 10$ (as was demonstrated in Section 3.1), and $r = 1$ are shown in Table 3.

Table 3: Expected test duration for r = 1 and s = 10.

Replicates (n)	D for $p = 10^{-9}$	D for $p = 10^{-7}$	D for $p = 10^{-5}$
1	114,155 yr	1,142 yr	11.4 yr
10	11,416 yr	114.2 yr	1.15 yr
100	1,142 yr	11.4 yr	42 days
1,000	114.2 yr	1.15 yr	4.2 days
10,000	11.4 yr	42 days	10 hours

From these estimates we can infer the number of replications n of the simulated testing environment that would be needed to obtain an experimental result within some time bound D, assuming the simulations execute an order of magnitude faster than real time. This analysis suggests that testing for the reliability of safety critical software with p near 10^{-7} may be feasible with modern, high performance computing systems, which have on the order of 10^4 general purpose cores with significant GPU support for simulating complex, physical environments. Even if the software application has a computationally active instruction sequences forcing $s \approx 1$, these numbers indicate that reliability of 10^{-6} would still be possible.

5 CONCLUSIONS

Simulation based testing of the type described here is not entirely new, with emulators playing a large role in testing embedded applications and mobile software. The new, emerging capability is for those

emulators to interact with detailed, dynamical models of the environment in a fast as possible simulation. When coupled with the availability of vast computational resources this approach to testing could have a transformative effect on software reliability by enabling the relatively inexpensive accumulation of vast numbers of testing hours in a simulated, but still relevant, operational environment.

Testing in the proposed manner is predicated on the availability of simulation models that represent the physical environment with sufficient fidelity. If adequate models are available from prior work on system design, then construction of the model does not impose additional costs during testing. Otherwise, the testing budget must include model construction in practical applications.

The specific types of emulation technology - hypervisors or computer system emulators - best suited to bringing operational software into a constructive simulation remains an open question. While an argument is made by Yoginath et al. (2015) to avoid solutions that synchronize the passage of real time in the emulator to simulation time in the model (see pg. 442, Section 1.3.2; their argument seems to implicitly include some forms of time dilation, such as discussed in (Grau et al. 2009)), this caution leaves space for a wide range of potential solutions.

Much ongoing research into the use of virtualization technologies to enable software testing, and particularly in regards to integrating virtualization into simulated environments, has focused on commercially relevant desktop and server hardware. This focus necessarily limits the capabilities of the simulation systems, which must use modern computing hardware to emulate modern computing hardware. Moreover, because of the tremendous complexity of modern computing hardware, research focuses on repurposing existing virtualization tools. Because these were not designed with simulation applications in mind, they impose technical constraints, such as timing errors when communicating between QEMU and the discrete event simulator that are described in Section 2.

We make two observations concerning these technical constraints. The first is that KVM appears to offer enough information about the state of the emulated microprocessor that run faster than real time simulations are possible, and this would be done just as it is with the *icount* mode (see the KVM documentation at https://www.kernel.org/doc/Documentation/virtual/kvm/api.txt). This information is not used by QEMU, likely because faster than real time simulations are not particularly relevant to a virtualization tool. Nonetheless, future enhancements of QEMU, or some other future tool intended specifically for software testing, could exploit this KVM feature.

The second observation is that for many safety critical systems, it may be possible to simulate the microprocessor in faster than real time. For example, avionics computers use hardware that is designed for high reliability, and this precludes some of the sophistication seen in modern, general purpose microprocessors. The Orion spacecraft, which began flight tests in 2014, has at the heart of its flight computer an IBM PowerPC 750FX (Whitwam, R. 2014). This computer processor has not appeared in commodity products since the early 2000's. It features a single core and an order of magnitude less computational power than the most modern microprocessors.

It is conceivable that a faster than real time simulation of the Orion flight computer could be constructed using a modern workstation computer. Coupled with relevant models governing the flight dynamics, it may be possible to experimentally demonstrate a given level of reliability for the flight control software. An approach like this is described by Shankar et al. (2014) for the Indian Mars Orbiter. Similar opportunities may exist for many, if not most, safety critical software systems.

ACKNOWLEDGMENTS

This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access

Plan (http://energy.gov/downloads/doe-public-access-plan). Research sponsored by the Laboratory Directed Research and Development Program of Oak Ridge National Laboratory.

REFERENCES

Butler, R. W., and G. B. Finelli. 1993. "The Infeasibility of Quantifying the Reliability of Life-critical Real-time Software". *IEEE Transactions on Software Engineering* 19(1):3–12.

Chiang, M.-C., T.-C. Yeh, and G.-F. Tseng. 2011. "A QEMU and SystemC Based Cycle-Accurate ISS for Performance Estimation on SoC Development". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30(4):593–606.

Grau, A., K. Herrmann, and K. Rothermel. 2009. "Efficient and Scalable Network Emulation Using Adaptive Virtual Time". In *2009 Proceedings of 18th International Conference on Computer Communications and Networks*, 1–6. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Gupta, D., K. V. Vishwanath, M. McNett, A. Vahdat, K. Yocum, A. Snoeren, and G. M. Voelker. 2011. "DieCast: Testing Distributed Systems with an Accurate Scale Model". *ACM Transactions on Computer Systems* 29(2):4–48.

Hu, X., and B. P. Zeigler. 2005. "Model Continuity in the Design of Dynamic Distributed Real-time Systems". *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans* 35(6):867–878.

IEEE Standards Association 2012. *IEEE Standard for Standard SystemC Language Reference Manual*. IEEE Computer Society. IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005).

Kurimoto, Y., Y. Fukutsuka, I. Taniguchi, and H. Tomiyama. 2013. "A Hardware/Software Cosimulator for Network-on-Chip". In *Proceedings of the 2013 International Soc Design Conference (ISOCC)*, 172–175. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Lee, H. W., D. Thuente, and M. L. Sichitiu. 2014. "Integrated Simulation and Emulation Using Adaptive Time Dilation". In *Proceedings of the 2nd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, 167–178. New York, New York: Association for Computing Machinery.

Montòn, M., J. Carrabina, and M. Burton. 2009. "Mixed Simulation Kernels for High Performance Virtual Platforms". In *2009 Forum on Specification Design Languages (FDL)*, 1–6. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Montòn, M., A. Portero, M. Moreno, B. Martinez, and J. Carrabina. 2007. "Mixed SW/SystemC SoC Emulation Framework". In *IEEE International Symposium on Industrial Electronics*, 2338–2341. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Mueller, W., and F. Pétrot. 2011. "QEMU and SystemC". In *1st International QEMU Users' Forum*. March 18th, Grenoble, France, 4.

Nutaro, J., O. Ozmen, J. Sanyal, D. Fugate, and T. Kuruganti. 2016. "Simulation Based Design and Testing of a Supervisory Controller for Reducing Peak Demand in Buildings". In *International High Performance Buildings Conference*. July 11th-14th, Purdue University, West Lafayette, Indiana, 1-7.

Nutaro, J. J. 2011. *Building Software for Simulation: Theory and Algorithms, with Applications in C++*. Hoboken, New Jersey: John Wiley & Sons.

Ozmen, O., J. J. Nutaro, J. Sanyal, and M. M. Olama. 2017. "Simulation-based Testing of Control Software". Technical Report ORNL/TM–2017/45, Oak Ridge National Laboratory, Oak Ridge, Tennessee.

Pantoquilho, M. 2010. "Challenges in Testing and Validating Operational Spacecraft Simulators". In *Proceedings of the 2010 Conference on Grand Challenges in Modeling & Simulation*, 165–172. Vista, California: Society for Modeling & Simulation International.

QEMU 2019. "QEMU Website". http://www.qemu.org, accessed 27th July.

Shankar, S. S., K. Desai, S. Dutta, R. R. Chetwani, M. Ravindra, and K. M. Bharadwaj. 2014. "Mission Critical Software Test Philosophy a SILS Based Approach in Indian Mars Orbiter Mission". In *2014 International Conference on Contemporary Computing and Informatics (IC3I)*, 414–419. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Weingärtner, E., F. Schmidt, T. Heer, and K. Wehrle. 2008. "Synchronized Network Emulation: Matching Prototypes with Complex Simulations". *SIGMETRICS Performance Evaluation Review* 36(2):58–63.

Whitwam, R. 2014. "NASAs Orion Spacecraft Runs on a 12 Year-old Single-core Processor from the iBook G3". https://www.geek.com/chips/nasas-orion-spacecraft-runs-on-a-12-year-old-single-core-processor-from-the-ibook-g3-1611132/, accessed 27th July.

Yoginath, S. B., K. S. Perumalla, and B. J. Henz. 2015. "Virtual Machine-based Simulation Platform for Mobile Ad-hoc Network-based Cyber Infrastructure". *The Journal of Defense Modeling and Simulation: Applications, Methodology, Technology* 12(4):439–456.

AUTHOR BIOGRAPHIES

JAMES NUTARO received his Ph.D. degree in Computer Engineering from the University of Arizona, Tucson, in 2003. He was a Systems Engineer with Raytheon Missile Systems and Northrop Grumman Information Systems, and since 2005 he has been a part of the research staff in the Computational Sciences and Engineering Division at the Oak Ridge National Laboratory, Oak Ridge, TN. His research interests include hybrid and discrete event simulation,simulation-based testing and model-based design of engineered systems, distributed and parallel simulation, and modeling methods. His email address is nutarojj@ornl.gov.

OZGUR OZMEN is a Research Scientist in the Computational Sciences and Engineering Division at the Oak Ridge National Laboratory, Oak Ridge, TN. His research interests are Complex Adaptive Systems, Agent-based Simulation, Model-based Engineering, and bridging the gap between Simulation and Artificial Intelligence. He holds an Industrial Engineering degree from Yildiz Technical University in Istanbul, Master of Engineering Management degree from Galatasaray University in Istanbul, MISE, and Ph.D. Degrees (both in Industrial and Systems Engineering) from Auburn University in Alabama. His email address is ozmeno@ornl.gov.