Scaling NS-3 DCE Experiments on Multi-Core Servers

David P. Wiggins, Leonid Veytser, Patricia Deutsch, and Bow-Nan Cheng
MIT Lincoln Laboratory
244 Wood Street
Lexington, MA 02420-9108
{david.wiggins, veytser, patricia.deutsch, bcheng}@ll.mit.edu

ABSTRACT
Direct Code Execution (DCE) is an NS-3 feature that enables ordinary executable programs to run largely unmodified on the nodes in an NS-3 simulation, enabling a single protocol implementation going from simulation to real hardware. Using the same implementation in all environments reduces development effort and makes behavior and results across environments more consistent and comparable. Although DCE has been used to evaluate smaller networks and verify protocol functionality, there has been little work scaling DCE to large number of nodes in wireless experiments. In this paper, we present our experiences with porting applications to DCE and using DCE to attempt to simulate large scale MANETs. Our results show that naively applying DCE to wireless simulations on multi-core servers can result in poor performance. With some minor configuration changes, however, significant speedup can be achieved.

CCS Concepts
• **Networks~Network simulations** • *Computer systems organization~Multicore architectures* • *Software and its engineering~Multithreading*

Keywords
DCE; NUMA; MANET

1. INTRODUCTION
The value of a simulation's results increases with its ability to predict behavior in the real world. For network protocols, that predictive power is often hampered by using different software implementations of the protocols in simulation vs. hardware. Two implementations of the same protocol will inevitably have different behavior. For example, they usually have different defects, or the implementers may have inconsistent interpretations of the protocol specification, or they may vary in performance characteristics. If the same implementation can be leveraged in both simulation and reality, these root causes of behavior mismatches can be eliminated, making the simulation results more valuable. As an added bonus, the effort to re-implement the protocol can be avoided.

Direct Code Execution (DCE) [2] is a feature of Network Simulator Version 3 (NS3)[1] that addresses this desire to use the same code both within and outside of simulation. It enables ordinary executable programs to run with little or no modification on the nodes in an NS-3 simulation. Such programs can send and receive network traffic via the normal socket application programming interface (API) that appears as traffic in the simulation. When run under DCE, a program that normally runs within its own process address is instead mapped to its own thread within the NS-3 address space. Consequently, a program that needs to run on all *n* nodes of a simulation causes DCE to start *n* additional threads.

We present our experiences with porting applications to DCE and using DCE on servers with substantial memory and processing resources in an attempt to simulate large-scale mobile ad-hoc networks (MANETs). While DCE has been used to test protocol correctness in limited wired scenarios, there is little work that attempts to scale DCE to large number of wireless nodes. We show that attempting to scale MANETs with DCE comes with significant challenges, some of which can be resolved with minor configuration changes and others that may require architectural changes.

In this paper, we describe the scenario under test, the hardware used for executing simulations, modifications that were necessary to applications and to DCE, the issues encountered when running DCE on a multi-core server, and our solutions to those issues. If applied naively, simulation performance with DCE in our target environment is surprisingly poor; a modern laptop computer outperforms the resource-rich server by a wide margin. The cause of this performance disparity and our mitigation strategy is explained. The key lessons learned are:

- Running a new program under DCE may require changes to the program, to DCE, or both.
- To run multiple simulations in parallel on one machine, one must make arrangements to ensure effective concurrent use of resources.
- DCE's use of threads interacts strongly with multi-core machines, resulting in severely degraded performance. Manual intervention is necessary to achieve better performance.

The rest of the paper is organized as follows: Section 2 describes related work. Section 3 describes the simulation scenario and hardware platform we used. Section 4 gives our observations about using DCE, while Section 5 discusses the observations. Finally, Section 6 concludes the paper with future work.

2. RELATED WORK

In recent years, NS-3 Direct Code Execution [2] has enabled higher fidelity comparisons of new algorithms against existing real-world implementations. By providing a framework to run deployed code including Linux kernel code in a simulation environment, increased realism and confidence in results can be obtained. Since its inception in early 2011, there have been numerous studies and demonstrations to leverage DCE in wired and limited wireless protocol development and verification.

In [2], the authors reproduce an existing experiment with Multipath TCP (MPTCP) using the same software in DCE. In the experiment, only two wireless links (LTE and Wi-Fi) are setup to examine MPTCP, resulting in limited scaling issues. In [12], a large 100 node wired powerline communication (PLC) network scenario was used to evaluate DLC-3000 protocols. In these tests, although a larger number of nodes were leveraged, all the tests were performed on a wired powerline network. In almost all cases of tests leveraging DCE, limited wireless nodes and/or wired networks were evaluated. In [18], large MANET networks up to 1000 nodes were simulated, but these did not use DCE. Although [11] demonstrates a MANET, it is also unclear whether significant performance evaluation was performed on the network.

In short, almost no literature exists that attempts to scale MANET experiments with native code and NS-3 DCE. MANET networks are uniquely difficult to scale due to the difficulty in partitioning the network to leverage optimized scheduling of simulation events. Almost all literature that scale NS-3 networks to large number of nodes (100,000+) [14][15][16] leverage NS-3 Message Passing Interface (MPI)[13] which enables splitting of the simulation into logical processes. Synchronization between logical processes is performed using various techniques. Unfortunately, MPI can only be leveraged on point-to-point links and is unsuitable for scaling wireless simulations.

In this paper, we attempt to leverage DCE to examine typical MANET routing protocols (OLSR and SMF) in a large scale MANET. Although our goal was to scale to hundreds of nodes, we observed several limitations. We present the lessons learned from attempting to scale MANETs in DCE.

3. TEST SCENARIO AND ENVIRONMENT

Our ultimate simulation goal was to obtain baseline network performance results for unicast and multicast traffic in wireless scenarios, using well-understood routing components. In particular, we were interested in testing the scalability of these protocols. This paper focuses on our experiences using NS3-DCE to obtain these baseline results. In the following subsections, we highlight the scenario and simulation server details that were used for our tests. All simulations used NS version 3.22 and DCE version 1.5.

3.1 Scenario Details

The specific scenario that this paper focuses on was a 47 node, mostly-connected, wireless ad-hoc network with mobility. The link data rate was 250 Kb/sec. The packet drop probability was 0.5, i.e., half of the packets were dropped at each hop. The simulation duration was 600 seconds. Unicast and multicast traffic started after 30 seconds and stopped 30 seconds before the end of the simulation in order to allow any residual packets to reach their destination. During the simulation, about 75K unicast packets and 93K multicast packets were sourced. The physical/MAC layer model was the NS-3 SimpleWireless model with some local enhancements.

Some of our simulations used different parameters, notably a larger number of nodes (125), a higher data rate, and lower packet drop probability, which resulted in a much larger number of packets for NS-3 to process. The intention was to scale up to several hundred nodes, but performance issues, some of which are described below, limited the simulations to less than a hundred nodes in order to achieve reasonable run-time. In the extreme, a single simulation took on the order of days to finish – certainly not what we considered reasonable. In order to isolate the performance issues, we chose the scenario described in the above paragraph because it finished in the least amount of time – on the order of hours rather than days. This shortened the time needed to run a simulation and analyze the results, enabling us to explore more possibilities and locate the issues more quickly.

The scenario used the Optimized Link State Routing Protocol (OLSR)[6] for unicast routing and Simplified Multicast Forwarding (SMF)[7] for multicast routing. To facilitate a future comparison against real hardware or against an emulator such as the Extendable Mobile Ad-hoc Network Emulator (EMANE)[20], DCE was used to run the Navy Research Lab (NRL) implementations (NRLOLSR[3] and NRLSMF[4]) of these protocols. In addition, NRL Multi-Generator (MGEN)[5] was executed using DCE to generate all the traffic for the scenario. We chose this suite of applications because it is a mature, portable collection of tools that work well together.

3.2 Simulation Server Details

We ran the simulations on a Dell® PowerEdge M520 blade server[8] running Ubuntu Linux 14.04. This machine had two Xeon E5-2450 processors providing 32 logical cores (physically 16 cores, but Intel® Hyper-Threading doubles this) and 128 GB of RAM organized as shown in Figure 1 [9].

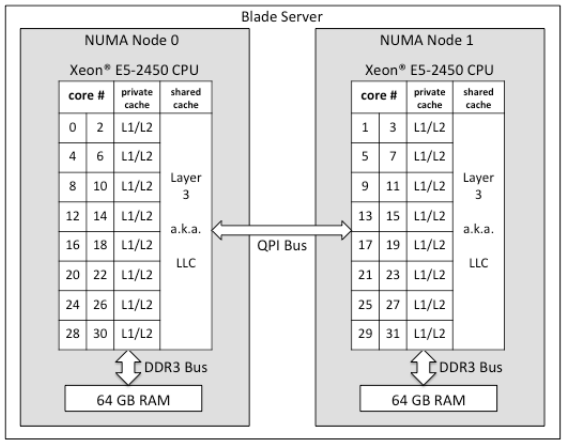

Figure 1. Blade server hardware organization.

The server's resources are divided evenly across two NUMA (Non-Uniform Memory Access) nodes. Each node's CPU is one physical chip, so this server has two CPU sockets. All of the cores can use all 128 GB of memory. However, core accesses to memory that resides on the same NUMA node as the core are faster than accesses to memory on a different NUMA node. This is because the latter access must travel two hops: the QPI (Quick Path Interconnect) bus to get to the other node, then DDR3 bus to

reach the memory. This disparity in memory access speeds is the hallmark of a NUMA architecture.[17]

Access to RAM is mediated through several levels of cache memory. Each pair of hyper-threaded cores, e.g. 0 and 2, share a level 1 (L1) and level 2 (L2) cache. All of the cores on a node share the level 3, or Last Level, cache (LLC). If a core wants to access data that is resident in another core's cache, for example a variable that is shared between threads, a cache coherency protocol comes into play to ensure a consistent view of the data across the cores. This is sometimes called a *ccNUMA (cache-coherent NUMA)* architecture, though the *cc* prefix is often omitted because nearly all modern NUMA architectures incorporate cache coherency.

4. DCE OBSERVATIONS

This section describes our experiences and observations stemming from using DCE with the scenario and hardware presented in Section 3.

4.1 Using NRL Applications with DCE

As mentioned in section 3.1, we chose NRLOLSR, NRLSMF, and MGEN as our routing and traffic generation tools to run under DCE. This section provides some details about these tools and discusses the effort that was necessary to make it possible to run them as DCE applications.

4.1.1 NRL Applications

MGEN is a traffic generator tool that provides the ability to send and receive TCP and UDP traffic as well as either unicast or multicast traffic. It provides a multitude of scriptable configurations to tune the traffic patterns based on your needs.

NRLOLSR is software that implements OLSR routing protocol based on RFC 3626[6]. It affects the routing of unicast traffic by injecting routes to the Linux kernel based on the OLSR protocol state.

NRLSMF is software that implements Simplified Multicast Forwarding based on RFC 6621[7]. It forwards multicast traffic by promiscuously listening on one or more MANET interfaces and forwarding received multicast traffic back out on these interfaces based on a set of rules. It can also be configured as a gateway to the local area network and forward multicast traffic from the LAN interface to the MANET interface and vice versa. Some of the forwarding rules include duplicate packet detection and time to live (TTL) checks. It also supports multiple forwarding methods. With classical flooding, every NRLSMF instance performs the forwarding. Alternatively, NRLSMF can work with an external program to perform more sophisticated forwarding. For example, NRLOLSR can be configured to control NRLSMF and only tell a subset of NRLSMF instances to forward based on Multipoint Relay (MPR) elections in OLSR. This inter-process communication between NRLOLSR and NRLSMF is implemented using named pipes.

All of the NRL tools that have been described use the Protean Protocol Prototyping Library (ProtoLib)[19]. It is a cross platform library that provides a set of classes and an overall framework for developing protocols.

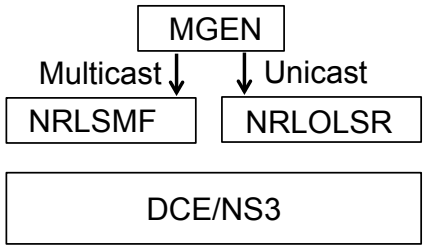

Figure 2. NRL tools used with DCE.

The NRL software works natively in the EMANE emulation environment, but we had to integrate these tools with NS-3 DCE to run over NS-3 wireless models.

Figure 2 depicts the how the NRL tools run on top of DCE and NS-3.

4.1.2 Modifications to DCE and Applications

To integrate the NRL software with DCE, several system calls that were missing under DCE had to be implemented. The system calls included:

- sched_get_priority
- sched_setscheduler
- chmod
- mkstemp64
- if_nameindex
- free_nameindex

NRL software also had to be modified. NRLSMF used *popen()* to spawn a command line *date* command to get the current time of day. This non-portable method does not work in DCE, so all of the *popen()* calls were replaced with *time()* and *localtime()* system calls that DCE already implemented. Additionally, we were only able to run NRLSMF in the classical flooding mode, where name pipe communication was not necessary. The inter-process communication between NRLOLSR and NRLSMF over named pipe never worked under DCE. Fixing this inter-process communication issue is a future work item.

We also ran into various crashes during the cleanup and destruction stage when the simulation exits. When various destructors in NRL software were called by DCE upon the scenario completion, closing various sockets and file handles caused crashes. Since this issue appeared after the scenario had already completed, we removed the offending calls to close the sockets to allow for the scenario to exit gracefully. Understanding the root cause of these crashes and providing a fix is also a future work item.

Finally, we were only able to run NRLSMF and NRLOLSR with Linux network stack. Both of the routing protocols require device names, specified as command line arguments, over which to perform their routing and forwarding. In the Linux network stack environment, devices have names with the naming convention of "sim[ifindex]." For example, the first device created is sim0, the second sim1 and so forth. Unfortunately, when using the NS-3 network stack this was not possible. The NS-3 devices have no concept of names, and there was no way to map NS-3 devices to the device names that NRLSMF and NRLOLSR expect. Fortunately, DCE's option to use the Linux network stack instead of the NS-3 stack enabled us to proceed with using the NRL software.

4.2 Running Parallel Simulations

Our simulation studies required sweeping through ranges of parameters, such as data rate, packet error rate, traffic load, and number of nodes, and running a simulation for each combination. To minimize the amount of time needed to complete all of the simulations, we planned to run multiple simulations at the same time on a blade server, since they had plentiful cores and memory. Our initial attempts to run parallel simulations led to mysterious crashes of the simulation executable. Eventually we determined that the crashes were caused by an implementation detail of DCE that interacted with the way we were running the simulation.

When a simulation that uses DCE starts, DCE must load the programs and related libraries (collectively, object files) into the NS-3 address space for execution. This task is handled by an ELF (Executable and Linkable Format) loader. The ELF loader may modify the object files as it loads them, for example to relocate embedded addresses to match the address at which they are actually being loaded. To save this relocation effort on future runs, DCE saves the modified programs and libraries in the directory `elf-cache` in the NS-3 directory tree.

When starting a batch of parallel simulations, our scripts used the same NS-3 tree for all of them, so they all used the same `elf-cache` directory. Moreover, all of the simulations started at approximately the same time. As a result, all of the simulations were overwriting the same files in the `elf-cache` directory at the same time, and the files were getting corrupted. It is not surprising that executing corrupted object files resulted in simulation crashes.

To work around this problem, we created a separate copy of the NS-3 directory tree for each parallel simulation so that the `elf-cache` directories would be unique to one simulation. This eliminated the file corruption so that the simulations no longer crashed.

4.3 Multi-core Performance Issues

In addition to running simulations in parallel, to get the most simulation throughput (simulations completed per unit time), optimizing the running time of the individual simulations was also of great interest. Our first indication that something unusual was happening came when we discovered that the same simulation ran significantly faster on a circa-2014 MacBook Pro laptop than it did on our high-performance blade server. The MacBook ran the simulation scenario described in section 3.1 in about 3.5 hours, while the blade server took about 12 hours. This was even more surprising because the MacBook was running the simulation inside a virtual machine (Ubuntu 14.04), while the blade server was running the same operating system directly on the hardware.

One of the most obvious differences between the laptop environment and the blade server was the filesystem that the simulation was using. On the laptop, it was a local disk. On the blade server, it was an NFS filesystem because all of the user home directories were NFS filesystems, and the NS-3 build resided in a user home directory. NFS operations can be much slower than a local disk.

DCE creates a directory per simulated node called `files-N` (N=node number) which serves as the root of a filesystem that can be used by DCE programs. These `files-N` directories are usually created in the NS-3 tree, in our case on an NFS filesystem. Logging for OLSR, SMF, and MGEN was initially very verbose, so the NFS filesystem was experiencing significant traffic. We modified DCE to store these `files-N` directories in /tmp/<username>/ `$DCE_INSTANCE_ID/`, moving them off of the NFS filesystem. `DCE_INSTANCE_ID` is an environment variable we introduced to keep parallel simulations segregated. Our scripts arrange for each simulation in a parallel set of simulations to have a unique value for `DCE_INSTANCE_ID` so their `files-N` directories remain separate. Redirecting the simulation disk I/O in this way had only a small impact on the simulation performance, so something else was causing the majority of the performance drop on the blade server.

Our investigation then turned to other straightforward measures including the following:

- We reduced the amount of logging and PCAP packet traces by changing log levels and disabling PCAP traces.
- We recompiled NS-3, DCE, and the applications (OLSR, SMF, MGEN) with optimization enabled.

However, none of these changes had a significant impact on the performance difference between the MacBook and the blade server.

We then focused on hardware differences. The Ubuntu VM on the MacBook had 1 core and 8 GB of memory, versus 32 cores and 128 GB on the blade server. Could having fewer resources actually make the simulation run faster? To test whether the critical resource was memory, we used `ulimit` to restrict the simulation to only 8 GB of memory on the blade server, but this had little effect, so we knew memory was not the issue.

We then turned our attention to the difference in the number of cores. Recall that DCE maps user programs to threads in the NS-3 address space. Our simulations had the main NS-3 thread doing most of the work plus three DCE threads (OLSR, SMF, and mgen) per node. For our 47-node scenario, the simulation had 142 active threads. We believed that our core-rich blade server would be a good match to run a process with this many threads. To verify this, we configured the blade server to match the laptop in number of cores by using the `taskset` command to confine the simulation to a single core. This was the key. Limiting the simulation to run on just one core on the blade server caused the simulation run time to plummet and to be on par with the MacBook.

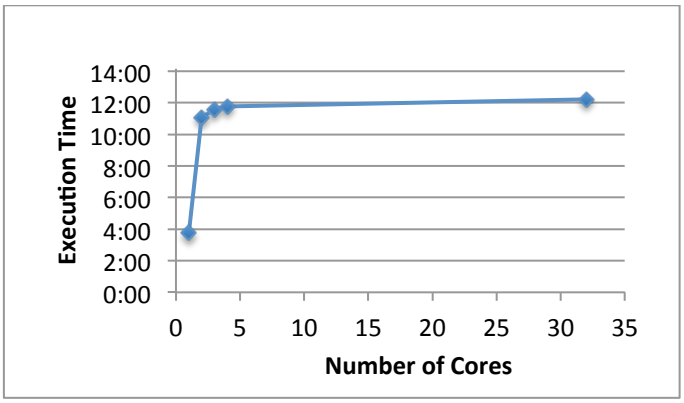

Figure 3. Simulation execution time vs. # of cores available.

To study this phenomenon in more detail, we ran the simulation with varying numbers of cores and measured the run time. To pin the simulation to a specific set of cores, we switched from using

`taskset` to `numactl` (with the `-physcpubind` option), as the latter is more flexible and seemed to establish the core assignments more reliably. The results are shown in Figure 3. Going from one to two cores, the simulation time jumped from 3h45m to 11h5m. Using additional cores resulted in much smaller increases in run time. When we placed no restrictions on core usage, the simulation time reached a maximum of 12h12m. We ran all of our remaining simulations using one core.

During these investigations, the `htop` program proved invaluable for quickly understanding how the cores of the blade server were being used. Figure 4 shows an `htop` screenshot taken when three parallel simulations were running, each assigned to one core. The bar graphs in the top half show cores 2, 3, and 4 pegged at 100% utilization.

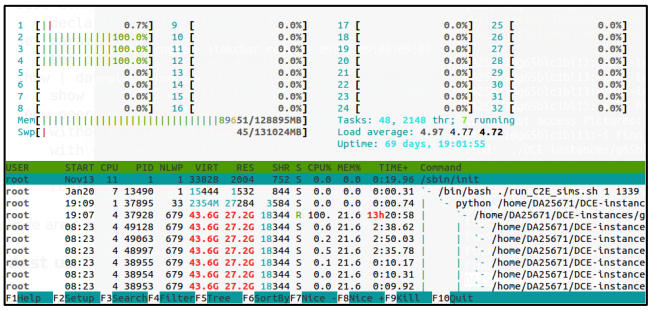

Figure 4. htop screenshot on the 32 core blade server.

Table 1 summarizes the utility programs we used to perform the experiments in this section that may be less familiar to the typical command-line user.

Table 1. Utility Programs for Multi-Core Systems

Name	Description
`taskset`	Set the CPU affinity of a process, which defines which core(s) the operating system should run the process on.
`numactl`	Show the NUMA topology of the machine, or control the NUMA scheduling and memory allocation behavior of a process. Much more flexible than `taskset`.
`htop`	List running processes/threads with a bar graph showing how busy each core is. It is useful for verifying that core allocations performed via `taskset` or `numactl` are actually in effect.

5. DISCUSSION

One possible explanation for the run time difference when going from one core to two or more cores is context switching. When the operating system decides to run a particular thread on a core, the execution state (mostly machine registers) of the previously running thread on that core must be saved, and the core must assume the new thread's state. In our simulation, where the number of threads is almost five times the number of available cores, context switching occurs frequently. It is very likely that a thread will run on a different core than it was run on the last time it was scheduled. As a result, some of the thread's data is likely to reside in the cache of a different core, so the cache coherency protocol will be invoked when the thread needs that data. This can be very expensive, especially if it happens repeatedly, and even more so if the cache that needs to be read is on a different NUMA node. However, if the thread runs on the same core as it did the last time it ran, there is a chance that its data will still be in one of that core's caches, and the cache coherency costs can be avoided. Note that if the entire simulation is running on one core, threads will always run on the same core. This could account for some of the performance increase when running on only one core.

Cache coherence could be coming into play in another way as well. If all of the DCE threads are accessing the same piece of data, particularly if many of those accesses are writes, then the cache coherency protocol will be constantly moving that data between cores. It is plausible that the DCE implementation has some housekeeping items, such as a lock, that exhibits such an access pattern. With only one core, this data will always be in the correct (and only) cache, so there would be no performance penalty.

6. CONCLUSIONS AND FUTURE WORK

Obtaining acceptable performance from a process with many threads on a multi-core machine is difficult. If multi-threading and NUMA issues can be considered during the design of the code that is to execute in this environment, then performance prospects are better. DCE's strategy of using a thread per user program requires the DCE user to take responsibility for mitigating the impact of these issues.

Because our focus was simply on finding a way to improve the speed of the simulations so we could complete our intended goal, we did not definitively determine the root cause of the inefficiencies but have made educated guesses about what it could be. It would be illuminating to determine the root cause of the slowdown when using multiple cores, starting with the conjectures described in section 5. A full understanding will require the use of more sophisticated tools such as `perf[21]` or Intel ® VTune ™ Amplifier[22], to expose behavior deeper inside the CPU, cache, and memory systems.

Opportunities to enhance DCE to function more efficiently on multi-core systems should be explored. The DCE implementation should be examined to determine what, if any, data is shared among all DCE threads. Such data should ideally be eliminated, or at least minimized. It may also be advantageous for DCE to actively manage the thread-to-core mappings.

Incorporating the small additions to DCE to enable running the NRL networking tools would lower the barrier to using DCE for those who wish to use those tools.

Because of the performance issues, ultimately we were only partially successful in using DCE to study wireless network scalability. Nevertheless, DCE is an impressive technical feat, and as it matures it is sure to become more useful to network researchers wishing to inject more realism into their simulations.

7. ACKNOWLEDGMENTS

This work is sponsored by the Defense Advanced Research Projects Agency (DARPA) under Air Force Contract # FA8721-05-C-0002. The views, opinions, and/or findings expressed are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

8. REFERENCES

[1] Network Simulator 3 (NS-3), www.nsnam.org

[2] H. Tazaki, F. Uarbani, E. Mancini, M. Lacage, D. Camara, T. Turletti, and W. Dabbous. 2013. Direct code execution: revisiting library OS architecture for reproducible network experiments. In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies* (CoNEXT '13). ACM, New York, NY, USA, 217-228. DOI=http://dx.doi.org/10.1145/2535372.2535374.

[3] NRL OLSR, www.nrl.navy.mil/itd/ncs/products/olsr

[4] NRL SMF, www.nrl.navy.mil/itd/ncs/products/smf

[5] NRL MGEN, www.nrl.navy.mil/itd/ncs/products/mgen

[6] Optimized Link State Routing Protocol, RFC 3626, www.ietf.org/rfc/rfc3626.txt

[7] Simplified Multicast Forwarding, RFC 6621, www.ietf.org/rfc/rfc6621.txt

[8] Dell® PowerEdge M520 Technical Guide, http://i.dell.com/sites/doccontent/shared-content/data-sheets/en/Documents/dell-poweredge-m520-technical-guide.pdf

[9] Intel® Xeon® Processor E5-2450 v2, http://ark.intel.com/products/75264/Intel-Xeon-Processor-E5-2450-v2-20M-Cache-2_50-GHz

[10] B. Sigoure. 2010. How long does it take to make a context switch?, http://blog.tsunanet.net/2010/11/how-long-does-it-take-to-make-context.html

[11] H. Tazaki, E. Mancini, D. Camara, T. Turletti, W. Dabbous . 2013. MSWIM demo abstract: direct code execution: increase simulation realism using unmodified real implementations. In *Proceedings of the 11th ACM international symposium on Mobility management and wireless access* (MobiWac '13). ACM, New York, NY, USA, 29-32. DOI=http://dx.doi.org/10.1145/2508222.2512837

[12] J. Deutschmann, A. Lehmann, J. Hampel, J. Huber. 2014. Network Simulation for Powerline Protocols with Direct Code Execution Applied to DLC-3000 SFN. In Proceedings of the 2014 IEEE International Conference on Smart Grid Communications (SmartGridComm 2014). IEEE, New York, NY, USA, 464-468. DOI=http://dx.doi.org/10.1109/SmartGridComm.2014.7007690

[13] Message Passing Interface, www.nsnam.org/docs/models/html/distributed.html.

[14] J. Pelkey and G. Riley . 2011. Distributed simulation with MPI in *ns-3*. In *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques* (SIMUTools '11). ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), ICST, Brussels, Belgium, Belgium, 410-414.

[15] K. Renard, C. Peri, and J. Clarke. 2012. A performance and scalability evaluation of the ns-3 distributed scheduler. In *Proceedings of the 5th International ICST Conference on Simulation Tools and Techniques* (SIMUTOOLS '12). ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), ICST, Brussels, Belgium, Belgium, 378-382.

[16] S. Nikolaev, P. Barnes, Jr., J. Brase, T. Canales, D. Jefferson, S. Smith, R. Soltz, and P. Scheibel. 2013. Performance of distributed *ns-3* network simulator. In *Proceedings of the 6th International ICST Conference on Simulation Tools and Techniques* (SimuTools '13). ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), ICST, Brussels, Belgium, Belgium, 17-23.

[17] U. Drepper, 2007. *What Every Programmer Should Know About Memory*. Red Hat, Inc.

[18] A. Quereilhac, D. Saucez, T. Turletti, and W. Dabbous. 2015. Automating ns-3 experimentation in multi-host scenarios. In *Proceedings of the 2015 Workshop on ns-3* (WNS3 '15). ACM, New York, NY, USA, 1-8. DOI=http://dx.doi.org/10.1145/2756509.2756513

[19] NRL Protolib, www.nrl.navy.mil/itd/ncs/products/protolib

[20] NRL EMANE, www.nrl.navy.mil/itd/ncs/products/emane

[21] perf wiki, https://perf.wiki.kernel.org/index.php/Main_Page

[22] Intel ® VTune ™ Amplifier, https://software.intel.com/en-us/intel-vtune-amplifier-xe