FINJ: A Fault Injection Tool for HPC Systems

Alessio Netti[1(✉)], Zeynep Kiziltan[1], Ozalp Babaoglu[1], Alina Sîrbu[2], Andrea Bartolini[3], and Andrea Borghesi[3]

[1] Department of Computer Science and Engineering, University of Bologna, Bologna, Italy
{alessio.netti,zeynep.kiziltan,ozalp.babaoglu}@unibo.it
[2] Department of Computer Science, University of Pisa, Pisa, Italy
alina.sirbu@unipi.it
[3] Department of Electrical, Electronic and Information Engineering, University of Bologna, Bologna, Italy
{a.bartolini,andrea.borghesi3}@unibo.it

Abstract. We present FINJ, a high-level fault injection tool for High-Performance Computing (HPC) systems, with a focus on the management of complex experiments. FINJ provides support for custom workloads and allows generation of anomalous conditions through the use of fault-triggering executable programs. FINJ can also be integrated seamlessly with most other lower-level fault injection tools, allowing users to create and monitor a variety of highly-complex and diverse fault conditions in HPC systems that would be difficult to recreate in practice. FINJ is suitable for experiments involving many, potentially interacting nodes, making it a very versatile design and evaluation tool.

Keywords: Exascale systems · Resiliency
Fault detection · Monitoring · Benchmarking · Open-source

1 Introduction

Motivation. High-Performance Computing (HPC) systems have become indispensable for economic growth and scientific progress in our modern society. As the performance of HPC systems increases, the value of the results they produce increases through higher-fidelity simulations, better predictive models and analysis of greater quantities of data. The resulting techniques, policy decisions and vastly-improved manufacturing processes in areas such as agriculture, engineering, transportation, materials, energy, health care, security and the environment are bound to impact most aspects of our lives. Today, HPC systems are also being used as fundamental "instruments" to achieve groundbreaking results in basic sciences ranging from particle physics to cosmology. Yet, many important problems in various fields remain unsolvable with current computational resources. *Exascale* HPC systems, capable of 10^{18} operations per second, are believed to

be essential for solving such problems [2]. Reaching exascale performance is the *moonshot* for modern HPC systems with many nations and companies engaged in an *arms race* towards achieving it.

Exascale systems, when they arrive, will come at a significant cost: scaling current technologies to exascale performance through massive parallelism will result in systems that have prohibitively-high levels of power consumption [17] and excessively-high failure rates [4]. Thus, to be usable in production environments with acceptable *Quality of Service* levels, exascale systems need to improve their power efficiency and resiliency by several orders of magnitude.

In our terminology, a *fault* is defined as an anomalous behavior at the software or hardware level that can lead to illegal system states (*errors*) and, in the worst case, to service interruptions (*failures*) [7]. In this paper, we limit our attention to improving the resiliency of HPC systems through the use of mechanisms for predicting, detecting and preventing errors and failures. An important technique in this endeavor is *fault injection*: the deliberate triggering of faults in a system so as to observe their behavior in a controlled environment, enable development of new prediction and response techniques and testing of existing ones [11]. For fault injection to be effective, dedicated tools are necessary, allowing users to trigger complex and realistic fault scenarios in a reproducible manner.

Related Work. Fault injection for prediction and detection purposes has been a topic of great interest in recent years. In [6,8,9,16], the authors employed software-based fault injection techniques to observe the behavior and performance variations of HPC systems in anomalous conditions, and to detect such faults using system performance metrics. However, while characterizing the fault-simulating programs that were used, these works do not focus on the tools used to inject and coordinate the faults themselves in the system.

Several studies have proposed fault injection tools with varying levels of abstraction. Calhoun et al. [3] devised a compiler-level fault injection tool focused on memory bit-flip errors, targeting HPC applications. De Bardeleben et al. [5] proposed a logic error-oriented fault injection tool. This tool is designed to inject faults in virtual machines, by exploiting emulated machine instructions through the open-source virtual machine and processor emulator (QEMU). Both works focus on low-level fault-specific tools and do not provide functionality for the injection of complex workloads, and for the collection of produced data, if any.

Stott et al. [15] proposed NFTAPE, a high-level and generic tool for fault injection. This tool is designed to be integrated with other fault injection tools and triggers at various levels, allowing for the automation of long and complex experiments. The tool however has aged considerably, and is not publicly available. A similar fault injection tool was proposed by Naughton et al. [14], however, to the best of our knowledge, it has never progressed past the prototype stage and is also not publicly available. Moreover, both tools require users to write a fair amount of wrapper and configuration code, resulting in a complex setup process. The Gremlins Python package[1] also supplies a high-level fault injector.

[1] https://github.com/toddlipcon/gremlins.

However, it does not support workload or data collection functionalities, and experiments on multiple nodes cannot be performed.

Joshi et al. [12] introduced the PREFAIL tool, which allows for the injection of failures at any code entry point in the underlying operating system. This tool, like NFTAPE, employs a coordinator process for the execution of complex experiments. It is targeted at a specific type of fault (code-level errors) and does not permit performing experiments focused on performance degradation and interference, among other fault types. Similarly, the tool proposed by Gunawi et al. [10], named FATE, allows the execution of long experiments; furthermore, it is focused on reproducing specific fault sequences, simulating real scenarios. Like PREFAIL, it is limited to a specific fault type, namely I/O errors, thus greatly limiting its scope.

Contributions. The main contribution of this paper is the design and implementation of FINJ, an easy-to-use open-source Python tool for fault injection targeted at HPC systems, with workload management capabilities. A relevant feature of FINJ is the possibility of seamless integration with other injection tools targeted at specific fault types, thus enabling users to coordinate faults from different sources and different system levels. By using FINJ's *workload* feature, users can also specify lists of applications to be executed and faults to be triggered on multiple nodes at specific times with specific durations. FINJ thus represents a high-level, flexible tool, enabling users to perform complex and reproducible experiments, aimed at revealing the complex relations that may exist between faults, application behavior and the system itself. FINJ is also extremely easy to use: it can be set up and executed in a matter of minutes, and does not require the writing of additional code in most of its usage scenarios. To the best of our knowledge, FINJ is the first portable, open-source tool that allows users to perform and control complex injection experiments, that can be integrated with heterogeneous fault types and that includes workload support, while retaining ease of use and a quick setup time.

Organization. The rest of the paper is structured as follows. In Sect. 2, we describe the FINJ architecture (Sect. 2.1), its components (Sect. 2.2) and their implementation (Sect. 2.3). In Sect. 3, we present a simple use case to show how FINJ can be deployed, while Sect. 4 concludes the paper.

2 FINJ Architecture

In this Section we discuss how fault injection is achieved in FINJ. We then present its architecture, together with some implementation details. Due to its portable and modular nature, customizing FINJ for different purposes is easy.

2.1 Architecture Overview

Fault injection in FINJ is achieved through *tasks* that are executed on target nodes: each task corresponds to a particular application, which can either be

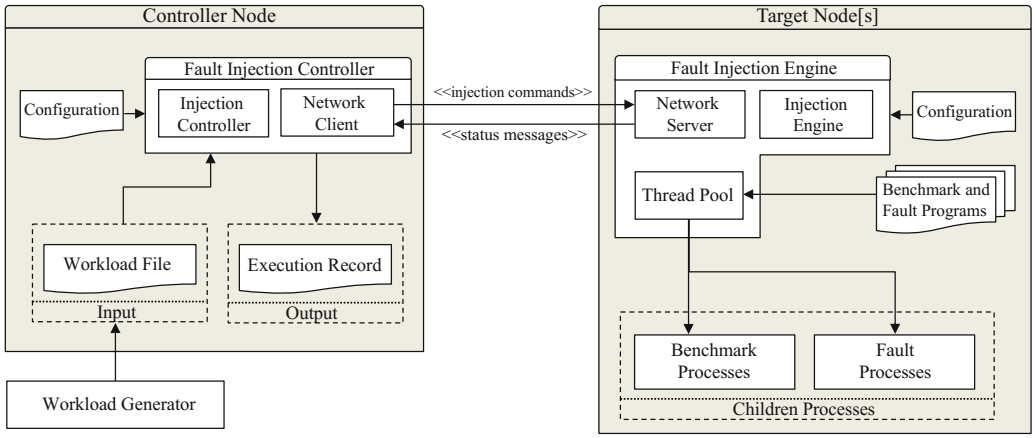

Fig. 1. Architecture of the FINJ tool showing the division between a controller node (left) and a target node (right).

a benchmark program or a fault-triggering program. As demonstrated in [15], this approach allows for the integration in FINJ of any low-level fault injection framework that can be triggered by using an executable program or a shell script. A task is defined by the following attributes:

- *args*: the full shell command required to run the selected task. The command must refer to an executable file that can be accessed from the target hosts;
- *timestamp*: the time in seconds at which the task must be started, relative to the starting time of the injection session;
- *duration*: the task's *maximum allowed* duration, expressed in seconds, after which it will be abruptly terminated. This duration can serve as an *exact* duration as well, with FINJ restarting the task if it finishes earlier, and terminating it if it lasts more. This behavior depends on the FINJ configuration (see Sect. 2.2). A duration of 0 implies that the task is always allowed to run until its termination;
- *isFault*: defines whether the task corresponds to a fault-triggering program, or to a benchmark application;
- *seqNum*: a sequence number used to uniquely identify the task;
- *cores*: the list of CPU cores that the task is allowed to use on target nodes, enforced through a *NUMA Control* policy [13]; this attribute is optional.

A set of tasks defines a *workload*, which is a succession of scheduled fault and benchmark executions at specific times, reproducing a realistic working environment for the fault injection process. A particular execution of a given workload then constitutes an *injection session*. Many fault programs are supplied with FINJ, allowing users to experiment with a variety of anomalies out-of-the-box.

FINJ consists of two basic components: a fault injection *controller*, and a fault injection *engine*. The two components correspond to processes that must be run on the nodes subject to injection experiments. Communication between them is achieved through TCP *sockets* using a simple message-based protocol. The high-level structure of the FINJ architecture is illustrated in Fig. 1.

FINJ Controller. The controller is the orchestrator of the injection process, and should be run on an external node that is not affected by the faults. The controller maintains connections to all nodes involved in the injection session, which run fault injection *engine* instances and whose addresses are specified by users when launching the program. Therefore, injection sessions can be performed on multiple nodes at the same time. The controller reads task entries from the selected workload: the reading process is incremental, and tasks are read in real-time a few minutes before their expected execution, according to their relative time-stamp. For each task the controller sends a command to all target hosts, instructing them to start the new task at the specified time. Finally, the controller collects all status messages produced by the target hosts, and stores them in a separate file for each host. These status messages are related to the start and termination of each single task, besides status changes in the host (for example, when connection is lost and re-established).

FINJ Engine. The engine is structured as a daemon, perpetually running on nodes that are expected to be subject to injection sessions. The engine waits for task commands to be received from remote controller instances. Engines can be connected to multiple controllers at the same time, and status messages will be sent to all of them. However, task commands are accepted from one controller at a time, which is defined as the *master* of the injection session. The engine manages received task commands by assigning them to a dedicated thread from a *pool*. The thread manages all aspects related to the execution of the task, such as spawning the necessary subprocesses and sending status messages to controllers when relevant events (such as the start or termination of the task) occur. Whenever a fault causes a target node to crash and reboot, controllers are able to re-establish and recover the previously running injection session, given that the engine is set up to be executed at boot time on the target node.

2.2 Components

FINJ is based on a highly modular architecture, and therefore it is very easy to customize single components in order to add or tune features.

Network. Engine and controller instances communicate through a network layer in the FINJ tool. Communication is achieved through a simple message-based protocol employing TCP sockets. This design choice is motivated by the fact that the volume of data sent during injection sessions is extremely low, while high reliability is a desirable quality. Users can still integrate their preferred transport method with little effort, thanks to FINJ's highly modular nature.

Specifically, a message *client* and *server* were implemented: clients are used by FINJ controllers in order to connect to servers hosted on FINJ engine instances. Messages can then be either *commands*, related to single tasks and imposed by controllers, or *status* messages, which are sent by engines and are related to status changes in their system. All messages are in the form of *dictionaries*. This component also handles resiliency features such as automatic

re-connection from clients to servers, since temporary connection losses are to be expected in a fault injection context.

Thread Pool. Task commands in FINJ engines are assigned to a thread in a pool as they are received: each thread manages all aspects of a task assigned to it. Specifically, the thread sleeps until the scheduled starting time of the task (according to its time-stamp); then, it spawns a subprocess running the specified task, and sends a message to all connected controllers to inform them of the event. At this point, the thread waits for the task's termination, depending on its duration and on the current configuration. Finally, the thread sends a new status message to all connected hosts informing them of the task's termination, and returns to sleep. The amount of threads in the pool, which is a configurable parameter, determines the maximum number of tasks that can be executed concurrently. Since threads in the pool are started only once during the engine's initialization, and wake up for minimal amounts of time when a task needs to be started or terminated, we expect their impact on performance to be negligible.

Input and Output. In FINJ, input and output of all data related to injection sessions are performed by controller instances, and are handled by *reader* and *writer* entities. By default, these employ the CSV format, which was chosen due to its extreme simplicity and generality, but they can be easily customized by users for other formats. *Input* in FINJ is constituted by *workload* files: as mentioned in Sect. 2.1, these files include one entry for each task that must be executed in the injection session. Using the CSV format makes workload files extremely readable, and manually writing workloads corresponding to highly specific test cases can be easily achieved as well. FINJ *output*, instead, is made up of two parts. The first is the *execution log*, which contains entries corresponding to status changes in the target node, namely the start and termination of tasks, errors that are encountered if any, and connection loss or recovery events. The second part of FINJ output is related to tasks: all output text written to the *stdout* or *stderr* channels during their execution, if any, is reported to controllers, and is stored in separate plain-text files in a directory alongside the main output file, each named according to the task's name and sequence number.

Configuration. The FINJ tool's runtime behavior is customizable by means of a configuration file. This file is in JSON format and includes several options that alter the behavior of either controller or engine instances. Among the basic options, it is possible to specify the listening TCP port for engine instances, and the list of addresses of target hosts, to which controller instances should connect at launch time. The latter is useful when injection sessions must be performed on large sets of nodes, whose addresses can be conveniently stored in a file. More complex options are also available: for instance, it is possible to define a series of commands corresponding to tasks that must be launched together with FINJ, and must be terminated with it. This option proves especially useful when users wish to set up monitoring frameworks, such as the *Lightweight Distributed*

Metric Service (LDMS) [1], to be launched together with FINJ in order to collect system performance metrics during injection sessions.

Workload Generation. While writing workload files manually is possible, this is time-consuming and not desirable for long injection sessions. Therefore, we implemented in FINJ a *workload generation* tool, which can be used to automatically generate workload files with certain statistical features, while trying to combine flexibility and ease of use. The workload generation process is controlled by three parameters: a maximum *time span* for the total duration of the workload expressed in seconds, a statistical distribution for the *duration* of tasks, and another one for their *inter-arrival* times. These distributions are separated in two sets, for fault and benchmark tasks, thus amounting to a total of four. They can be either specified analytically by the user or can be fitted from real data, thus reproducing realistic behavior.

A workload is composed as a series of fault and benchmark tasks that are selected from a list of possible shell commands. To control the composition of workloads, users can optionally associate to each command a probability for its selection during the generation process, and a list of CPU cores for its execution, as explained in Sect. 2.1. By default, commands are picked uniformly. Having defined its parameters, the workload generation process is then fairly simple: tasks are randomly generated in order to achieve statistical features close to those specified as input, and are written to an output CSV file, until the maximum imposed time span is reached. Alongside the full workload, a *probe* file is also produced: this workload file contains one entry for each task type, all with a short fixed duration, and represents a lightweight workload version. This file can be used during the setup phase to test the correct configuration of the system, making sure that all tasks are correctly found and executed on the target hosts, without having to run the entire heavy workload.

2.3 Implementation

FINJ is implemented in Python, an object-oriented, high-level interpreted programming language[2], and can be used on all major operating systems. All FINJ dependencies are included in the Python distribution, and the only optional external dependency is the *scipy* package, which is needed for the workload generation functionality. The source code is publicly available on GitHub[3] under the MIT license, together with its documentation, usage examples and several fault-triggering programs. FINJ works on Python versions 3.4 and above.

In Fig. 2 we illustrate the class diagram for the FINJ tool. The *engine* and *controller* entities are respectively represented by the *InjectorEngine* and *InjectorController* classes. Users can instantiate these classes and start injection sessions directly, by using the *listen* method to put the engine in listening mode, and the *inject* method of the controller, which allows to start the injection session

[2] https://www.python.org/events/python-events/.
[3] https://github.com/AlessioNetti/fault_injector.

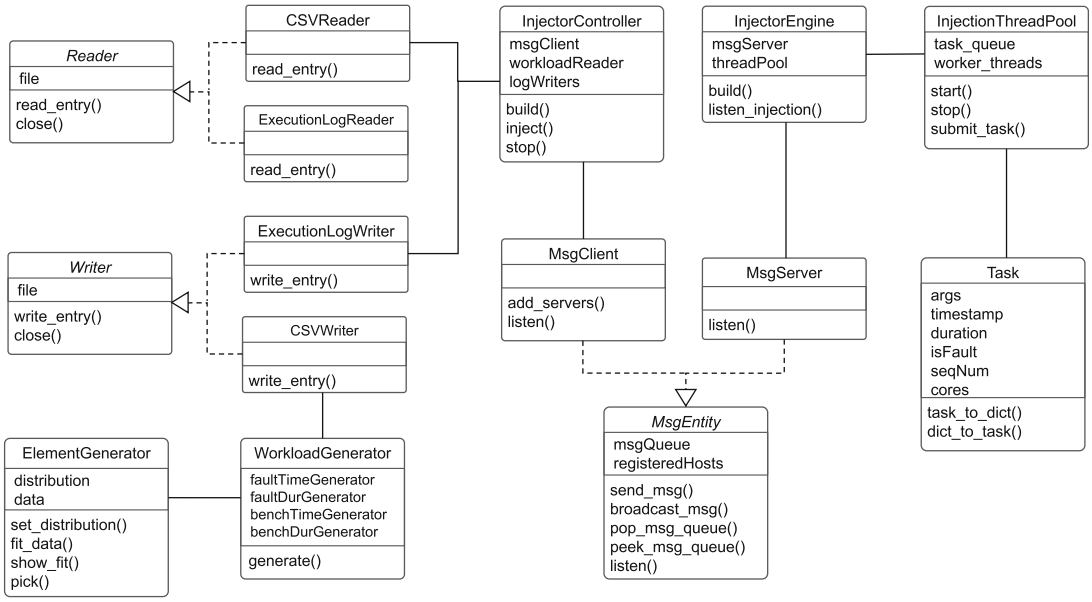

Fig. 2. Class diagram of the FINJ tool.

itself. However, scripts are supplied with FINJ to create controller and engine instances from a command-line interface, simplifying the process. This method will be discussed in Sect. 3. The *InjectionThreadPool* class, instead, supplies the thread pool implementation used to execute and manage tasks.

The network layer of the tool is represented by the *MsgClient* and *MsgServer* classes, which implement the message and queue-based client and server used for communication. Both classes are implementations of the *MsgEntity* abstract class, which provides the interface for sending and receiving messages, and implements the basic mechanisms that regulate the access to the underlying queue.

Input and output are instead handled by the *Reader* and *Writer* abstract classes and their implementations: *CSVReader* and *CSVWriter* handle the reading and writing of workload files, while *ExecutionLogReader* and *ExecutionLogWriter* handle execution logs generated by injection sessions. Since these classes are all implementations of abstract interfaces, it is easy for users to customize them for different formats. Tasks are modeled by the *Task* class that contains all attributes specified in Sect. 2.1.

Lastly, access to the workload generator is provided through the *WorkloadGenerator* class, which is the interface used to set up and start the generation process. This class is backed by the *ElementGenerator* class, which offers basic functionality for fitting data and generating random values. This class acts as a wrapper on scipy's *rv_continuous* class, which generates random variables.

3 Using FINJ

In this Section we demonstrate the flow of execution of FINJ through a concrete example carried out on a real HPC node and provide insight on its overhead.

```
timestamp;duration;seqNum;isFault;cores;args
0;1723;1;False;0-7;./hpl lininput
355;244;2;True;6;sudo ./cpufreq 258
914;291;3;True;4;./leak 316
```

Fig. 3. A sample CSV workload that can be used with FINJ.

3.1 Sample Execution

In this Section we will consider a sample fault injection session carried out using FINJ. The employed workload file, named *sample.csv*, is illustrated in Fig. 3. The test was carried out on one node of an HPC system equipped with two Intel Xeon E5-2630 v3 CPUs, each with 8 cores, 128 GB of RAM, and running CentOS 7.3. The *finj_engine* and *finj_controller* Python scripts are supplied with FINJ to start *engine* and *controller* instances respectively. Their usage is explained on the GitHub repository for the tool, together with all configuration options.

In this workload, the first task is the Intel Distribution[4] for the well-known *High-Performance Linpack* (HPL) benchmark, optimized for Intel Xeon CPUs. This task starts at time 0 in the workload, and has a maximum allowed duration of 30 min. The following two tasks are fault-triggering programs: *cpufreq* uses the Intel P-State driver in the Linux kernel[5] to dynamically reduce the maximum allowed CPU frequency, emulating performance degradation, while *leak* [16] creates a memory leak in the system, eventually using all available RAM. The cpufreq program requires appropriate permissions, so that users can access the files controlling Linux CPU governors. The HPL benchmark was run with 8 threads, pinned on the first 8 cores of the machine, while the cpufreq and leak tasks were forced to run on cores 6 and 4 respectively. Also note that the tasks must be available at the specified path on the systems running the FINJ engine, which in this case is relative to the location of the launching script.

Having defined the workload, the injection engine and controller must be started. Using the default configuration, and supposing that the test must be performed locally, this can be accomplished with the two following commands:

```
python finj_engine -p 30000 &
python finj_controller -w sample.csv -a localhost:30000
```

In the code above, the *-p* argument indicates the listening TCP port for the engine instance. The *-a* argument is instead the list of engine addresses to which the controller should connect, and *-w* is the path of the CSV workload file to be injected. The controller instance will then connect to the engine and start executing the workload, storing all output in a unique CSV file for each target host. When this process is finished, the controller terminates. The output CSV files for our example have the format shown in Fig. 4: each entry represents a status change event, which in this case is the start or termination of tasks, and

[4] https://software.intel.com/en-us/mkl-windows-developer-guide-overview-of-the-intel-distribution-for-linpack-benchmark.

[5] https://www.kernel.org/doc/Documentation/cpu-freq/intel-pstate.txt.

```
timestamp;type;args;seqNum;duration;isFault;cores;error
1529172604;command_session_s;None;None;None;None;None;None
1529172624;status_start;./hpl lininput;1;1723;False;0-7;None
1529172979;status_start;sudo ./cpufreq 258;2;258;True;6;None
1529173237;status_end;sudo ./cpufreq 258;2;258;True;6;None
1529173538;status_start;./leak 316;3;316;True;4;None
1529173855;status_end;./leak 316;3;316;True;4;None
1529174347;status_end;./hpl lininput;1;1723;False;0-7;None
1529174348;command_session_e;None;None;None;None;None;None
```

Fig. 4. A sample output file produced by FINJ after an injection session for the workload specified in Fig. 3.

is flagged with its absolute time-stamp on the target host. In addition, we also find an *error* field, detailing possible errors that were encountered. Note that the file is opened and closed by session *start* and *end* entries: the presence of these ensures that the injection process did not encounter errors and that the entire workload was processed successfully. It can be clearly seen from this experiment how easily a FINJ experiment can be configured and started on multiple cores.

At this point, the data generated by FINJ can be easily compared with other data, for example performance metrics collected through a monitoring framework, in order to better understand the system's behavior under faults. For this test, we used the LDMS framework [1] to collect performance metrics on the target host at each second, for the duration of the injection session. In Fig. 5 we show the total RAM usage and the CPU frequency of core 0. The benchmark's profile is simple, showing a constant CPU frequency while RAM usage slowly increases as the application performs tests on increasing matrix sizes. The effect of our fault programs, marked in gray, can be clearly observed in the system: the *cpufreq* fault causes a sudden drop in CPU frequency, resulting in reduced performance and longer computation times, while the *leak* fault causes a steady, linear increase in RAM usage. Even though saturation of the available RAM is not reached, this peculiar behavior can be used for prediction purposes.

3.2 Overhead of FINJ

We also performed tests in order to evaluate the overhead that FINJ may introduce. To do so, we employed the same system used in Sect. 3.1 together with the HPL benchmark, this time configured to use all 16 cores of the machine. We run the HPL benchmark 20 times directly, and then repeated the same process by using a FINJ workload. FINJ was once again instantiated locally. In both conditions the HPL benchmark scored an average running time of roughly 320 seconds, therefore leading us to conclude that the impact of FINJ on running applications is negligible, as expected from the implementation.

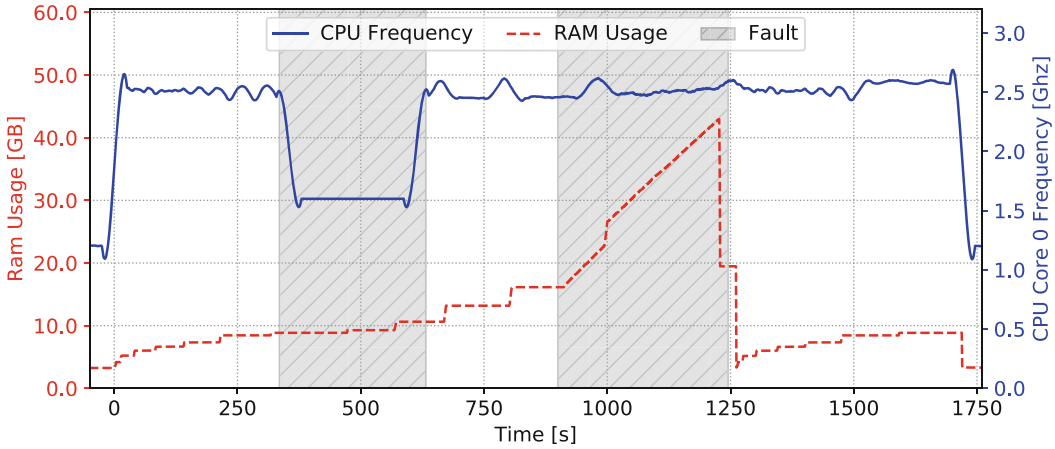

Fig. 5. CPU Frequency and RAM Usage, as monitored on the target system during the sample injection session.

4 Conclusions

We have presented FINJ, a high-level, easy-to-use tool for fault injection and monitoring in HPC systems. FINJ allows for the automation of complex experiments, and for reproducing anomalous behaviors in a deterministic, simple way. FINJ is open-source and implemented in Python, an object-oriented interpreted programming language available on all major operating systems, and has no dependencies for its core operation. This, together with the simplicity of its command-line interface, makes the deployment of FINJ on large-scale systems trivial. Since FINJ is based on the use of tasks, which are external executable programs, users can integrate the tool with any existing lower-level fault injection framework that can be triggered in such way, and ranging from the application level to the kernel, or even hardware level. The use of workloads in FINJ also allows to reproduce complex, specific fault conditions in HPC systems, and to reliably perform experiments involving multiple nodes at the same time.

As future work, we plan to perform scalability studies on the FINJ tool, by deploying it on a large-scale HPC environment. We have already performed extensive testing on the system presented in Sect. 3 with excellent preliminary results. Also, we plan to implement the ability to build workloads in which the order of tasks is defined by *causal* relationships rather than time-stamps, which might simplify the triggering of extremely specific anomalous states in a given system. We will also integrate multiple network transport methods to choose from besides TCP, so as to extend the range of systems FINJ can be applied to.

Acknowledgements. A. Netti has been supported by a research fellowship from the *Oprecomp-Open Transprecision Computing* project. A. Sîrbu has been partially funded by the EU project *SoBigData Research Infrastructure—Big Data and Social Mining Ecosystem* (grant agreement 654024).

References

1. Agelastos, A., et al.: The lightweight distributed metric service: a scalable infrastructure for continuous monitoring of large scale computing systems and applications. In: Proceedings of SC 2014, pp. 154–165. IEEE (2014)
2. Ashby, S., Beckman, P., Chen, J., Colella, P., Collins, B., Crawford, D., et al.: The opportunities and challenges of exascale computing. In: Summary Report of the Advanced Scientific Computing Advisory Committee (ASCAC) Subcommittee, pp. 1–77 (2010)
3. Calhoun, J., Olson, L., Snir, M.: FlipIt: an LLVM based fault injector for HPC. In: Lopes, L., et al. (eds.) Euro-Par 2014. LNCS, vol. 8805, pp. 547–558. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-14325-5_47
4. Cappello, F., Geist, A., Gropp, W., Kale, S., Kramer, B., Snir, M.: Toward exascale resilience: 2014 update. Supercomput. Front. Innovations **1**(1), 5–28 (2014)
5. DeBardeleben, N., Blanchard, S., Guan, Q., Zhang, Z., Fu, S.: Experimental framework for injecting logic errors in a virtual machine to profile applications for soft error resilience. In: Alexander, M., et al. (eds.) Euro-Par 2011. LNCS, vol. 7156, pp. 282–291. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29740-3_32
6. Ferreira, K.B., Bridges, P., Brightwell, R.: Characterizing application sensitivity to OS interference using kernel-level noise injection. In: Proceedings of SC 2008, p. 19. IEEE Press (2008)
7. Gainaru, A., Cappello, F.: Errors and faults. In: Herault, T., Robert, Y. (eds.) Fault-Tolerance Techniques for High-Performance Computing. CCN, pp. 89–144. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-20943-2_2
8. Guan, Q., Chiu, C.C., Fu, S.: CDA: a cloud dependability analysis framework for characterizing system dependability in cloud computing infrastructures. In: Proceedings of PRDC 2012, pp. 11–20. IEEE (2012)
9. Guan, Q., Fu, S.: Adaptive anomaly identification by exploring metric subspace in cloud computing infrastructures. In: Proceedings of SRDS 2013, pp. 205–214. IEEE (2013)
10. Gunawi, H.S., et al.: FATE and DESTINI: a framework for cloud recovery testing. In: Proceedings of NSDI 2011, p. 239 (2011)
11. Hsueh, M.C., Tsai, T.K., Iyer, R.K.: Fault injection techniques and tools. Computer **30**(4), 75–82 (1997)
12. Joshi, P., Gunawi, H.S., Sen, K.: PREFAIL: a programmable tool for multiple-failure injection. In: ACM SIGPLAN Notices, vol. 46, pp. 171–188. ACM (2011)
13. Lameter, C.: Numa (non-uniform memory access): an overview. Queue **11**(7), 40 (2013)
14. Naughton, T., Bland, W., Vallee, G., Engelmann, C., Scott, S.L.: Fault injection framework for system resilience evaluation: fake faults for finding future failures. In: Proceedings of Resilience 2009, pp. 23–28. ACM (2009)
15. Stott, D.T., Floering, B., Burke, D., Kalbarczpk, Z., Iyer, R.K.: NFTAPE: a framework for assessing dependability in distributed systems with lightweight fault injectors. In: Proceedings of IPDS 2000, pp. 91–100. IEEE (2000)

16. Tuncer, O., et al.: Diagnosing performance variations in HPC applications using machine learning. In: Kunkel, J.M., Yokota, R., Balaji, P., Keyes, D. (eds.) ISC 2017. LNCS, vol. 10266, pp. 355–373. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-58667-0_19
17. Villa, O., Johnson, D.R., O'connor, M., Bolotin, E., Nellans, D., Luitjens, J., et al.: Scaling the power wall: a path to exascale. In: Proceedings of SC 2014, pp. 830–841. IEEE (2014)