Readme of the Antarex Dataset

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

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

Abstract. The Antarex dataset contains trace data collected from an homonymous experimental HPC system located at ETH Zurich while it was subjected to fault injections. The dataset is publicly available for use by the community. In this document we give an overview of the dataset, by describing the experimental set up associated with data acquisition and discussing the features extracted from the dataset.

1 Dataset Overview

In order to acquire data, we executed benchmark applications and at the same time injected faults in the system at specific times via dedicated programs, so as to trigger anomalies in the behaviour of the applications. One type of data in the dataset refers to a series of CSV files, each containing a set of system performance metrics sampled through an HPC monitoring framework. Another type refers to the log files detailing the status of the system (i.e. currently running benchmark applications or injected fault programs) at each time point in the dataset. Such a structure enables researchers to perform a wide range of studies on the dataset. Moreover, since we collected the dataset by streaming continuous data, any study based on it will easily be reproducible on a real HPC system, in an online way.
The dataset is divided in two parts. The first part includes only the CPU and memory-related benchmark applications and fault programs, while the second is strictly hard drive-related. We executed each part in both single-core and multi-core variants. In the former, we executed all benchmark applications and fault programs on one specific core with one thread. In the latter, conversely, we executed benchmark applications with multiple threads on 8 of the 16 cores of the system, and executed fault programs freely on any of them. This structure resulted in 4 blocks of nearly 20GB of data in total, each block being obtained at different execution times, during an acquisition period of 32 days. The dataset

Table 1. A summary of the structure for the Antarex dataset.

Dataset Block	**Type**	**Parallel**	**Duration**	**Benchmark Programs**	**Fault Programs**
Block I	CPU-Mem	No	12 days	DGEMM[4], HPCC[5], STREAM[6], HPL[7]	leak, memeater, ddot, dial, cpufreq, pagefail
Block III		Yes			
Block II	Hard Drive	No	4 days	IOZone[8], Bonnie++[9]	ioerr, copy
Block IV		Yes			

structure is summarized in Table 1. The related benchmark applications and fault programs will be explained in the following subsections.

2 Experimental Setup for Data Acquisition

The Antarex HPC node used for data acquisition consists of two Intel Xeon E5-2630 v3 CPUs, 128GB of RAM, a Seagate ST1000NM0055-1V4 1TB hard drive and runs the CentOS 7.3 operating system. The node has a default Tier-1 computing system configuration. In order to schedule the execution of the benchmark applications and to inject faults, we used the FINJ tool [5] in a Python 3.4 environment, with its fault-injecting engine running on the target machine itself, and its orchestrating controller running on a remote host. In order to collect performance metrics from the target system for the duration of the experiment, we used the Lightweight Distributed Metric Service (LDMS) framework [1]. Like FINJ, the sampler component of LDMS was running on the target node, while the collector component was running on a remote host. We configured LDMS to sample a variety of metrics at each second, which come from the following seven different plug-ins:

1. *meminfo* collects general information on RAM usage;
2. *perfevent* collects CPU performance counters;
3. *procinterrupts* collects information on hardware and software interrupts;
4. *procdiskstats* collects statistics about hard drive usage;
5. *procsensors* collects metrics about CPU temperature and frequency;
6. *procstat* collects general metrics about CPU usage;
7. *vmstat* collects information about virtual memory usage.

This configuration resulted in a total of 2094 metrics collected at each second. Some of the metrics are system-wide, and describe the status of the system as a whole, others instead are core-specific and describe the status of a specific CPU core. Since there are 16 cores in our system, these metrics will have 16 instances as well, one for each core.

In order to minimize noise and bias in the sampled data, we chose to analyze, execute benchmarks and inject faults into only 8 of the 16 cores

[4] https://lanl.gov/projects/crossroads/benchmarks-performance-analysis.php
[5] https://icl.cs.utk.edu/hpcc/
[6] https://www.cs.virginia.edu/stream/
[7] https://software.intel.com/en-us/articles/intel-mkl-benchmarks-suite
[8] https://www.iozone.org
[9] https://www.coker.com.au/bonnie++/

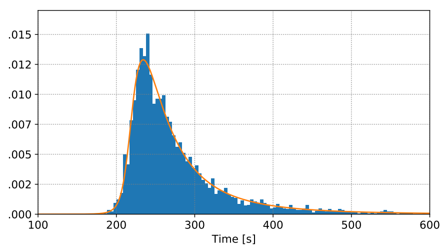
available in the system, and therefore used only one CPU. On the other CPU of the system, instead, we executed the FINJ and LDMS tools, which rendered their CPU overhead negligible.

3 Features of the Dataset

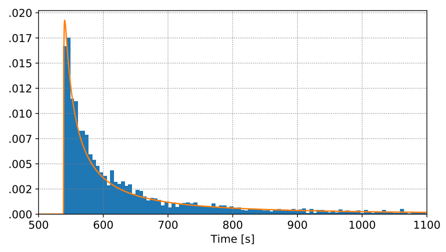
The FINJ tool orchestrates the execution of benchmark applications and the injection of faults by means of a workload file, which contains a list of benchmark and fault-triggering tasks to be executed at certain times, on certain cores, for certain durations [5]. For this purpose, we used several FINJ workload files, which were generated using FINJ's built-in workload generator, one for each block of the dataset.

Workload Files. We used two statistical distributions in the workload generator to create the durations and inter-arrival times of the benchmark and fault-triggering tasks. We define the inter-arrival time as the interval between the start of two consecutive tasks. The benchmark tasks are characterized by rather simple duration and inter-arrival features. By using normal distributions, we achieved that 75% of the dataset's duration is spent running benchmark applications. This resulted in regular benchmark tasks, having an average duration of 30 minutes, and average inter-arrival times of nearly 40 minutes.
The fault-triggering tasks are modeled in a more complex way. In order to achieve a realistic behavior, we chose to generate faults in the workload using statistical distributions fitted from real historical data, rather than specifying them analytically. For this purpose, we used the Grid5000 trace available on Fault Trace Archive[10], which includes the host failure records of the Grid5000 large-scale cluster [2] belonging to the period of May 2005 to November 2006. We extracted from this trace the inter-arrival times of the host failures. Such data was then scaled and shifted to obtain an average of 10 minutes, while retaining the shape of the distribution. We then fitted the data using an exponentiated Weibull distribution, which is commonly used to characterize failure inter-arrival times [3]. To model durations, we extracted for all hosts the time intervals between successive *absent* and *alive* records, which respectively characterize host failure and recovery events. We then fitted a Johnson SU distribution over a cluster of the data present at the 5 minutes point, which required no alteration in the original data. This particular type of distribution was chosen because of the quality of the fitting.
In Figure 1, we show the histograms for the durations (a) and inter-arrival times (b) of the fault tasks in the workload files, together with the original distributions fitted from the Grid5000 data. We observe that the histograms differ slightly at the peaks, compared to the respective reference distributions. This is because the workload generator is allowed to manipulate the durations and inter-arrival times to ensure that faults cannot overlap in time.

[10] http://fta.scem.uws.edu.au/



(a) Histogram of fault durations.



(b) Histogram of fault inter-arrival times.

Fig. 1. Histograms for fault durations (a) and fault inter-arrival times (b) in the Antarex dataset, compared to the PDFs of the Grid5000 fitted data.

Benchmark Applications. We used a series of well-known benchmark applications to load the Antarex HPC node while acquiring the dataset, stressing different parts of the system and providing a diverse environment for fault injection. Since we limit our analysis to a single machine, we use versions of the benchmarks that rely on shared-memory parallelism, for example through the OpenMP library. The benchmark applications are listed in Table 1 and are the following:

1. *DGEMM* measures matrix-to-matrix multiplication performance;
2. *HPC Challenge (HPCC)* is a collection of benchmarks that stress both the CPU and memory bandwidth of an HPC system;
3. *Intel distribution for High-Performance Linpack (HPL)* measures performance in solving a system of linear equations;
4. *STREAM* measures a system's memory bandwidth;
5. *Bonnie++* measures HDD read-write performance;
6. *IOZone* measures HDD read-write performance.

Fault Programs. We now discuss the fault programs that we implemented and used to reproduce anomalous conditions in the analyzed HPC system, which are available at the FINJ Github repository [5]. As in [6], each fault program can operate in a high or low-intensity mode, thus doubling the number of possible fault conditions. The programs, together with the generated 8 distinct faults and their effects, are the following:

1. *leak* periodically allocates 16MB arrays which are never released. In low-intensity mode, 4MB arrays are allocated [6]. This program produces a *memory leak* fault, which leads to memory fragmentation and severe system slowdown when memory saturation is reached;
2. *memeater* allocates a 36MB array which is filled with integers. The size of the array is then periodically increased and new elements are filled in. The application restarts after 10 iterations. In low-intensity mode, an 18MB array is used [6]. This program produces a *memory interference* fault by saturating memory bandwidth, resulting in degraded performance for running applications;

3. *ddot* repeatedly calculates the dot product between two equal-size matrices. The sizes of the matrices change periodically between 0.9, 5 and 10 times the cache's size. In low-intensity mode, the size of the matrices is halved [6]. This program produces a *CPU and cache interference* fault, resulting in degraded performance for all applications running on the same CPU as the program;
4. *dial* repeatedly generates random floating-point numbers and performs numerical operations over them. In low-intensity mode, the program sleeps for 0.5 seconds for each second of operation [6]. This program produces an *ALU interference* fault, resulting in degraded performance for applications running on the same core as the program;
5. *cpufreq* decreases the maximum allowed CPU frequency by 50% of its original value through the Linux Intel P-State driver[11]. In low-intensity mode, the maximum frequency is reduced by 30%. This program simulates a *system misconfiguration* or *failing CPU* fault, resulting in degraded performance for running applications;
6. *pagefail* makes any page allocation request fail with 50% probability, by using the Linux kernel's fault injection framework[12]. In low-intensity mode, page allocations fail with 25% probability. This program simulates a *system misconfiguration* or *hardware malfunction* fault, causing performance degradation and stalling of running applications;
7. *ioerr* triggers errors upon hard-drive I/O operations, again using the Linux kernel's fault injection framework. One out of 500 I/O operations fails with 20% probability in high-intensity mode, and with 10% probability in low-intensity mode. This program simulates a *failing hard drive* fault, causing degraded performance for I/O-bound applications, as well as potential errors and crashes;
8. *copy* repeatedly writes and then reads back a 400MB file from a hard drive. After such a cycle, the program sleeps for 2 seconds. In low-intensity mode, a 200MB file is used [4]. This program simulates an *I/O interference* or *failing hard drive* fault by saturating I/O bandwidth, and results in degraded performance for I/O-bound applications. Unlike ioerr, copy does not cause any I/O operations to fail and cause errors, but only slows them down, thus reproducing a different anomalous condition.

The faults triggered by our programs can be grouped in three categories according to their nature. The *interference* faults (i.e. leak, memeater, ddot, dial and copy) occur when orphan processes are left running in the system, saturating resources and slowing down the other processes. The *misconfiguration* faults occur when a component's behavior is outside of its specification, due to a misconfiguration by the users or administrators (i.e. cpufreq). Finally, the *hardware* faults are related to physical components in the system that are about to fail, and trigger various kinds of errors (i.e. pagefail or ioerr). We note that some faults may belong to

[11] https://www.kernel.org/doc/Documentation/cpu-freq/intel-pstate.txt
[12] https://www.kernel.org/doc/Documentation/fault-injection/fault-injection.txt

multiple categories, as they can be triggered by different factors in the system.

References

1. Agelastos, A., Allan, B., Brandt, J., Cassella, P., Enos, J., Fullop, J., Gentile, A., Monk, S., Naksinehaboon, N., Ogden, J., et al.: The lightweight distributed metric service: a scalable infrastructure for continuous monitoring of large scale computing systems and applications. In: Proc. of SC 2014. pp. 154–165. IEEE (2014)
2. Bolze, R., Cappello, F., Caron, E., Daydé, M., Desprez, F., Jeannot, E., et al.: Grid'5000: A large scale and highly reconfigurable experimental grid testbed. The International Journal of High Performance Computing Applications 20(4), 481–494 (2006)
3. Gainaru, A., Cappello, F.: Errors and faults. In: Fault-Tolerance Techniques for High-Performance Computing, pp. 89–144. Springer (2015)
4. Guan, Q., Fu, S.: Adaptive anomaly identification by exploring metric subspace in cloud computing infrastructures. In: Proc. of SRDS 2013. pp. 205–214. IEEE (2013)
5. Netti, A., Kiziltan, Z., Babaoglu, O., Sirbu, A., Bartolini, A., Borghesi, A.: FINJ: A fault injection tool for HPC systems. In: Proc. of Resilience Workshop 2018. Springer (2018), `https://github.com/AlessioNetti/fault_injector`
6. Tuncer, O., Ates, E., Zhang, Y., Turk, A., Brandt, J., Leung, V.J., et al.: Online diagnosis of performance variation in hpc systems using machine learning. IEEE Transactions on Parallel and Distributed Systems (2018)