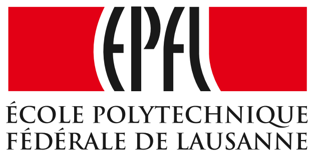
IX Open-source version 1.0 – Deployment and Evaluation Guide

http://github.com/ix-project

École Polytechnique Fédérale de Lausanne



George Prekas[1]
Adam Belay[2]
Mia Primorac[1]
Ana Klimovic[2]
Samuel Grossman[2]
Marios Kogias[1]
Bernard Gütermann[1]
Christos Kozyrakis[1,2]
Edouard Bugnion[1]

[1]: EPFL, [2]: Stanford University

May 27, 2016

IX Open-source version 1.0 – Deployment and Evaluation Guide

George Prekas Adam Belay Mia Primorac Ana Klimovic
Samuel Grossman Marios Kogias Bernard Gütermann Christos Kozyrakis
Edouard Bugnion

https://github.com/ix-project

Abstract

This Technical Report provides the deployment and evaluation guide of the IX dataplane operating system, as of its first open-source release on May 27, 2016. To facilitate the reproduction of our results, we include in this report the precise steps needed to install, deploy and configure IX and its workloads. We reproduce all benchmarks previously published in two peer-reviewed publications at OSDI '14 [3] and SoCC '15 [6] using this up-to-date, open-source code base.

1 Introduction

The IX dataplane operating system project started in 2013 as a collaboration between researchers at Stanford and EPFL. The design and implementation of the dataplane, which combines low latency and high throughput, was published at OSDI in 2014 [3]. The design and implementation of the control plane, which additionally ensures high system efficiency, was published at SoCC in 2015 [6].

Like all software projects, IX is in constant evolution. In recent months, the focus was to increase the robustness, debuggability and extensibility of the system.

As we release the code base in open-source for the first time, this report documents the four necessary components required to reproduce our research results:

- **the IX software:** this includes instructions to build and deploy IX, including its dependencies: Dune [2] and DPDK [4];
- **the benchmarks:** the full set of benchmarks that were used in prior publications.
- **the experimental setup:** this includes a description of the hardware used for those experiments as well as the complete description of the host Linux configuration.
- **the experimental results:** a complete re-evaluation of IX on all benchmarks, performed with the open-source code base. The results are generally consistent with the published numbers.

This work was funded by DARPA CRASH (under contract #N66001-10-2-4088), a Google research grant, the Stanford Experimental Datacenter Lab, the Microsoft-EPFL Joint Research Center, NSF grant CNS-1422088, and a VMware grant. George Prekas was supported by a Google Graduate Research Fellowship and Adam Belay by a VMware Graduate Fellowship.

2 Deploying IX v1.0

The upstream version of IX is available at

```
https://github.com/ix-project/ix
```

For normal development, simply clone and work off the `master` branch. To reproduce the behavior of IX version 1.0, and the experiments in this report, checkout the tag `v1.0`.

IX depends on three additional open-source projects, which are automatically downloaded via a provided script:

- Dune: IX requires a fork of Dune [2], available at `https://github.com/ix-project/dune`. The version used in this paper has commit hash `34a94ebbd1b09c84e870396302e4864f500cbba2`. Dune's upstream is at `http://dune.scs.stanford.edu/`.
- DPDK: IX uses Intel's DPDK framework version 16.04.
- PCI DMA management device driver: we have developed a management device driver that reliably enables and disables DMA for PCI devices – this driver is required to avoid memory corruption after using DPDK[1].

Additionally, a wiki with more detailed and up to date information on IX is available at `https://github.com/ix-project/ix/wiki`

2.1 Prerequisites

Currently IX has been successfully tested on Ubuntu 15.10 and 16.04 LTS running the Linux kernel version 4.2 and 4.4 respectively.

Currently, IX supports the following NICs:

- `Intel 82599`
- `Intel X520`
- `Intel X540`

Finally, IX requires the `libconfig` and `libnuma` libraries. You should use your package manager to download the above-mentioned libraries before attempting to build IX.

2.2 Build IX

The following steps download IX, its dependencies, the necessary libraries, and build them, on an APT-based Linux distribution:

```
git clone git@github.com:ix-project/ix
cd ix
# [OPTIONAL] git checkout v1.0
./deps/fetch-deps.sh
sudo chmod +r /boot/System.map-`uname -r`
make -sj64 -C deps/dune
make -sj64 -C deps/pcidma
make -sj64 -C deps/dpdk config T=x86_64-native-linuxapp-gcc
make -sj64 -C deps/dpdk
sudo apt-get install libconfig-dev libnuma-dev
make -sj64
```

The resulting executable files are

- `ix/cp/ixcp.py` – the IX control plane
- `ix/dp/ix` – the IX dataplane kernel

2.3 Configuration options

In order to successfully run the IX dataplane, you need to adjust the IX configuration to match your hardware and network setup. Typically, IX reads the `ix.conf` configuration file from the current working directory. A sample configuration file `ix.conf.sample` is provided as a configuration basis. You can also supply your own configuration file path using the `-c` command line argument. The available configuration parameters are:

[1] bug report: http://dpdk.org/ml/archives/users/2016-March/000340.html

host_addr	IP address and netmask that will be assigned to the adapter once it comes up in CIDR notation (e.g., 192.168.0.2/24)
gateway_addr	default gateway IP address (e.g., 192.168.0.1)
port	TCP port to bind for IX to listen on (e.g., 1234)
devices	A list of PCI device ID of NICs in BDF form (e.g., 0000:01:00.0)
cpu	A list of CPUs to run on (e.g., 0)
batch	maximum batch size of received packets to process (e.g., 64)
loader_path	path to the dynamic loader (e.g. /lib64/ld-linux-x86-64.so.2)

2.4 Environment configuration

To run IX, you must remove the kernel NIC driver and install the necessary kernel modules. Also, IX uses exclusively large pages (2MB), so you need to reserve a number of them before executing IX. IX takes over the whole NIC. The assumption here is that one has a second NIC for Linux or has terminal access to the server. If this is not the case, refer to the instructions below to run IX on a Virtual Function in order to multiplex the network card. To quickly deploy the correct IX environment, follow these steps:

```
cp ix.conf.sample ix.conf
# modify at least host_addr, gateway_addr, devices, and cpu
sudo sh -c ' \
  for i in /sys/devices/system/node/node*/hugepages/hugepages-2048kB/\
      nr_hugepages; do \
    echo 4096 > $i; \
  done'
sudo modprobe -r ixgbe
sudo insmod deps/dune/kern/dune.ko
sudo insmod deps/pcidma/pcidma.ko
```

Optionally, IX is compatible with PCI Virtual Functions on SR-IOV capable network cards. This allows sharing of the same NIC with Linux. This setup allows for both Linux and IX network stacks to be multiplexed on the same network card at a price of a slight performance hit especially on older NICs. To set up the environment for running IX on a PCI Virtual Function:

```
sudo modprobe ixgbe
PCI_DEVICE="$(basename "$(readlink "/sys/class/net/$IFACE/device")")"
sudo sh -c 'echo 1 > /sys/bus/pci/devices/$PCI_DEVICE/sriov_numvfs'
sudo ifconfig $VIRTUAL_IFACE up
sudo modprobe -r ixgbevf
```

where `$IFACE` is the logical name assigned to your IX-compatible network device (e.g., `eth0`), and `$VIRTUAL_IFACE` is the logical name assigned to the VF (e.g., `enp1s16`).

Finally, to undo the above steps and reload the default Linux network stack :

```
sudo modprobe ixgbe
sudo rmmod dune
sudo rmmod pcidma
sudo sh -c 'echo 0 > /sys/bus/pci/devices/$PCI_DEVICE/sriov_numvfs'
```

```
sudo modprobe -r ixgbevf
sudo sh -c ' \
  for i in /sys/devices/system/node/node*/hugepages/hugepages-2048kB/\
      nr_hugepages; do \
    echo 0 > $i; \
  done'
sudo ifconfig $IFACE up
```

2.5 Running the echo server

The IX `echoserver` program simply listens on a TCP port and echoes input back to the client once it has received 'n' bytes of content. To start it:

```
sudo ./dp/ix -- ./apps/echoserver 4
```

Or, alternatively :

```
sudo ./dp/ix -c <path_to_your_ix.conf_file> -- ./apps/echoserver 4
```

To test it, use another machine and open a TCP connection to the IP address and port specified in `ix.conf`, e.g.:

```
nc -vv <HOSTNAME> <PORT>
```

3 Benchmarks

The current distribution of IX includes three applications that demonstrate its functionality and measure different aspects of its performance: echoserver, NetPIPE and memcached.

3.1 Echoserver

Echoserver is a simple application that waits to receive a TCP payload of specified size and, then, sends it back to the sender. The echoserver is available on the IX tree at `https://github.com/ix-project/ix/blob/master/apps/echoserver.c`.

The echoserver application allows us to run two simple but useful performance benchmarks: (1) the throughput behavior of IX when scaling the number of connections with a fixed message size; and (2) the throughput behavior for variable message size with a fixed number of connections. (1) is available in the ix-bench tree as `bench_connscaling.py` while (2) is available in the tree as `bench_short.py`.

3.2 NetPIPE

NetPIPE is an application that uses a simple series of ping-pong tests over a range of message sizes to provide a complete measure of the performance of a network. It bounces messages of increasing size between two hosts across a network. Message sizes are chosen at regular intervals and with slight perturbations to provide a complete evaluation of the communication system. Each data point involves many ping-pong tests to provide an accurate timing. While we did some minor modifications to the original NetPIPE code, it essentially follows the same principles. The NetPIPE application is available at `https://github.com/ix-project/netpipe` while the upstream source is available at `http://bitspjoule.org/netpipe/code/NetPIPE-3.7.1.tar.gz`.

The NetPIPE application provides the ability to run benchmarks similar to `bench_short.py` for latency measurements, but with a different set of parameters. Because NetPIPE has not been ported to IX, we use the echoclient/echoserver on the IX network stack to generate NetPIPE workloads and compare these with the results of the NetPIPE application on the Linux network stack. This benchmark is available in the ix-bench tree as `bench_pingpong.py`.

3.3 Memcached

Memcached is an in-memory key-value store for chunks of arbitrary data (strings, objects). We use mutilate [5] to generate the dataset and to perform the actual benchmark. We forked memcached from version `1.4.18` and applied some minor changes to the source code. The memcached application is available at `https://github.com/ix-project/memcached` while the upstream source is available at `https://github.com/memcached/memcached`.

Memcached allows us to run synthetic benchmarks against datasets and workloads similar to the ones observed in production deployments. Our benchmarks follow the following pattern: multiple clients running multiple threads generate concurrent requests against a single instance of memcached running on IX. We recreate the request distributions observed at Facebook and reported by Atikoglu et al [1] using mutilate [5], a distributed memcached load generator available at `https://github.com/ix-project/mutilate`. There are three memcached benchmarks available in the ix-bench tree as `bench_memcached.py`, `bench_memcached_dynamic.py`, and `bench_memcached_pareto.py`.

4 Experimental setup

Our experimental setup consists of a cluster of 17 clients and one server connected by a low-latency 10 GbE switch. For the NetPIPE benchmark we use 2 hosts, for the 10 Gbps benchmarks we use 10 clients to generate load and 1 additional client to measure latency (if needed by the benchmark), and for the 40 Gbps benchmarks we use 17 clients. The client machines are a mix of Xeon E5-2637 @ 3.5 Ghz and Xeon E5-2650 @ 2.6 Ghz. The server is a dual Xeon E5-2665 @ 2.4 Ghz with 256 GB of DRAM. Each client and server socket has 8 cores and 16 hyperthreads. All machines are configured with one Intel x520 10GbE NIC (82599EB chipset). IX has exclusive access to the NIC (not using VFs). Our baseline configuration in each machine is Ubuntu 15.10, running the Linux kernel version 4.2. Although the server has two sockets, the foreground and background applications run on a single processor to avoid any NUMA effects. We run the control plane on the otherwise empty second socket, but do not account for its energy draw. For the 40 Gbps benchmarks, we used 4×10 Gbps links configured in `balance-xor` bonding mode.

The following modifications to operating system level parameters were applied prior to running the benchmarks:

1. Increase per-process system-wide resources limits. Particularly, we make sure that processes (1) can lock enough pages in the main memory and (2) have access to enough file descriptors to meet some of the benchmarks requirements. These limits can be retrieved using the (1) `ulimit -l` and (2) `ulimit -n` commands, and can be edited through the `limits.conf` file. In our experimental setup, we use (1) MEMLOCK=2097152 and (2) NOFILE=262144. *Note that these values are provided indicatively and may not meet the resource allocation requirements of any other configuration than our own.*

2. Prevent the Linux kernel from managing huge pages allocation. By default, the kernel automatically merges normal pages into huge pages and the opposite, which introduces variability in the benchmarks. In our environment, we disabled this feature using

   `sudo sh -c 'echo never > /sys/kernel/mm/transparent_hugepage/enabled'`

3. Disable TCP SYN cookies. To effectively monitor connection failures in a timely fashion, we disable TCP SYN cookies. This allows us to make sure that all connections are effectively alive on the server side. In our environment, we disabled this feature using

   `sudo sysctl net.ipv4.tcp_syncookies=0`

4. Disable TurboBoost and set the CPU to the maximum available. In our environment :

```
for i in /sys/devices/system/cpu/cpu*/cpufreq; do
    sudo sh -c "echo userspace > $i/scaling_governor"
    sudo sh -c "echo $MAX_FREQ > $i/scaling_setspeed"
done
```

   where `$MAX_FREQ` is the maximum non-TurboBoost CPU frequency.

Configuration	Minimum latency @99th pct		RPS for SLA: < 500µs @99th pct	
	OSDI	1.0	OSDI	1.0
ETC-Linux	94µs	65µs	550K	898K
ETC-IX	45µs	34µs	1550K	4186K
USR-Linux	85µs	66µs	500K	902K
USR-IX	32µs	33µs	1800K	5817K

Table 1 – Unloaded latency and maximum RPS for a given service-level agreement for the memcache workloads `ETC` and `USR`.

5 Results

We hereby present the results with the open source version of IX of the benchmarks published at OSDI '14 [3]. The figure map is the following:

Figure/Table	OSDI	this TR
NetPIPE	Fig. 2	Fig. 1
Multi-core scalability	Fig. 3	Fig. 2
Connection scalability	Fig. 4	Fig. 3
Connection scalability (hardware counters)		Fig. 4
Memcached USR and ETC	Table 1	Table 1
Memcached USR and ETC	Fig. 5	Fig. 5
Batch sensitivity	Fig. 6	Fig. 6

For SoCC '15 [6], the figure map is the following:

Figure/Table	SoCC	this TR
Pareto	Fig. 2	Fig. 7
Pareto/DVFS only	Fig. 3	Fig. 8
Energy Proportionality	Fig. 6	Fig. 9, 10, 11 (left)
Workload consolidation	Fig. 7	Fig. 9, 10, 11 (right)
Power savings and cons. gains	Table 1	Table 2
Flow Migration statistics	Table 2	Table 3

Table 1 shows that the performance has substantially improved between OSDI and v1.0 for both Linux and IX. This due to a combination of optimizations in the newer Linux kernel (from 3.16 to 4.2), in the configuration of the memcached application, and in IX itself.

	Smooth		Step		Sine+noise	
	SoCC	v1.0	SoCC	v1.0	SoCC	v1.0
Energy Proportionality (W)						
Max. power	91	92	92	93	94	94
Measured	42 (-54%)	43 (-54%)	48 (-48%)	46 (-51%)	53 (-44%)	51 (-46%)
Pareto bound	39 (-57%)	38 (-59%)	41 (-55%)	38 (-59%)	45 (-52%)	43 (-54%)
Server consolidation opportunity (% of peak)						
Pareto bound	50%	53%	47%	51%	39%	44%
Measured	46%	47%	39%	43%	32%	35%

Table 2 – Energy Proportionality and Consolidation gains.

		avg	95th pct.	max.	stddev
add core	prepare (µs)	119	212	10082	653
	wait (µs)	113	475	1018	158
	rpc (µs)	102	238	378	75
	deferred (µs)	125	460	2534	283
	total (µs)	462	1227	12804	927
	# packets	83	285	2753	280
remove core	prepare (µs)	23	49	312	25
	wait (µs)	33	106	176	31
	rpc (µs)	12	27	48	7
	deferred (µs)	16	43	82	11
	total (µs)	86	154	370	40
	# packets	3	9	25	3

Table 3 – Breakdown of flow group migration measured during the six benchmarks.

References

[1] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload Analysis of a Large-Scale Key-Value Store. In *Proceedings of the 2012 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 53–64, 2012.

[2] A. Belay, A. Bittau, A. J. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis. Dune: Safe User-level Access to Privileged CPU Features. In *Proceedings of the 10th Symposium on Operating System Design and Implementation (OSDI)*, pages 335–348, 2012.

[3] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proceedings of the 11th Symposium on Operating System Design and Implementation (OSDI)*, pages 49–65, 2014.

[4] Intel Corp. Intel DPDK: Data Plane Development Kit. `http://dpdk.org/`, 2014.

[5] J. Leverich and C. Kozyrakis. Reconciling High Server Utilization and Sub-Millisecond Quality-of-Service. In *Proceedings of the 2014 EuroSys Conference*, pages 4:1–4:14, 2014.

[6] G. Prekas, M. Primorac, A. Belay, C. Kozyrakis, and E. Bugnion. Energy proportionality and workload consolidation for latency-critical applications. In *Proceedings of the 2015 ACM Symposium on Cloud Computing (SOCC)*, pages 342–355, 2015.

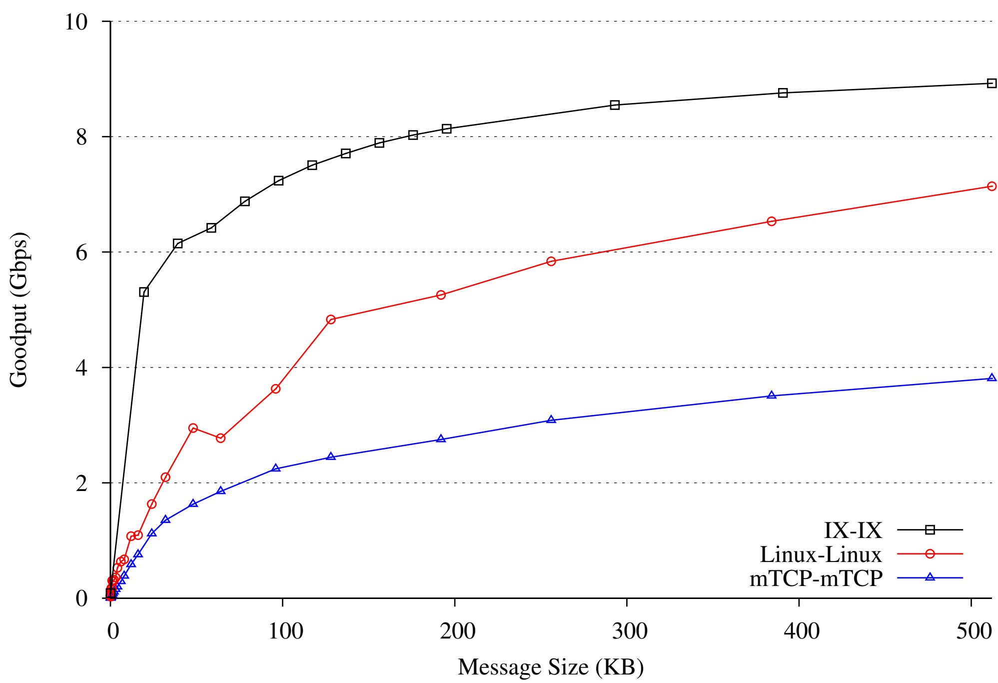
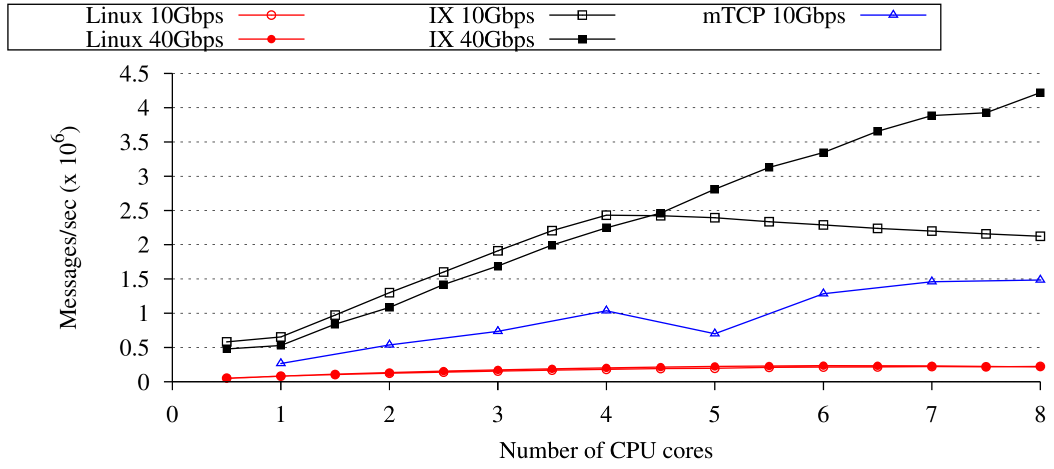
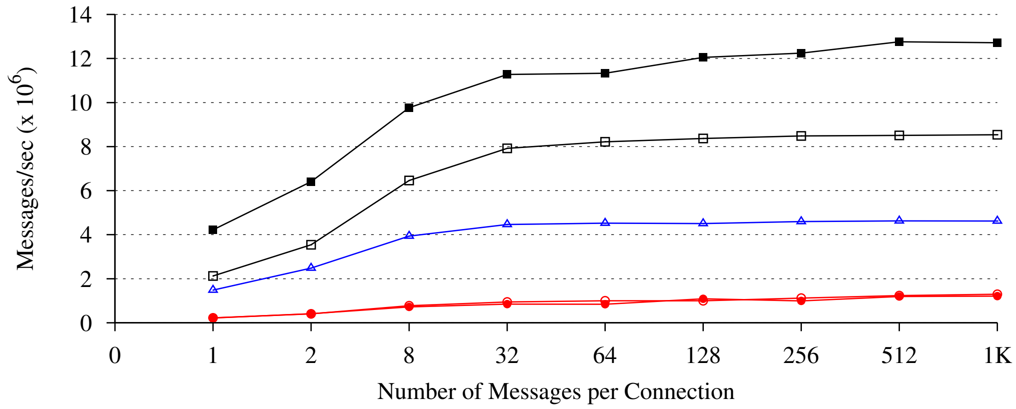


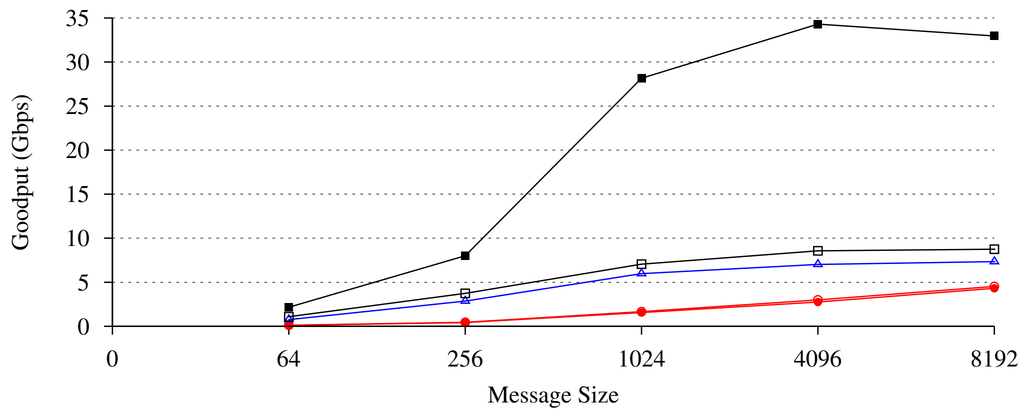
Figure 1 – NetPIPE performance for varying message sizes and system software configurations.



(a) Multi-core scalability (n=1, s=64B)



(b) n round-trips per connection. (s=64B)



(c) Different message sizes s (n=1)

Figure 2 – Multi-core scalability and high connection churn for 10GbE and 4x10GbE setups. In (a), half steps indicate hyperthreads.

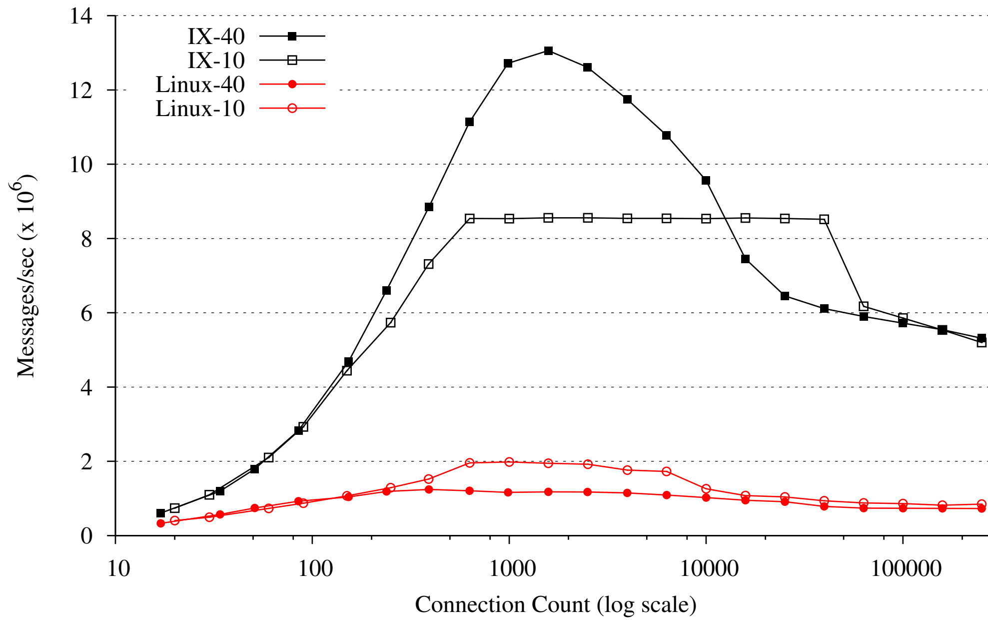


Figure 3 – Throughput for varying established connections

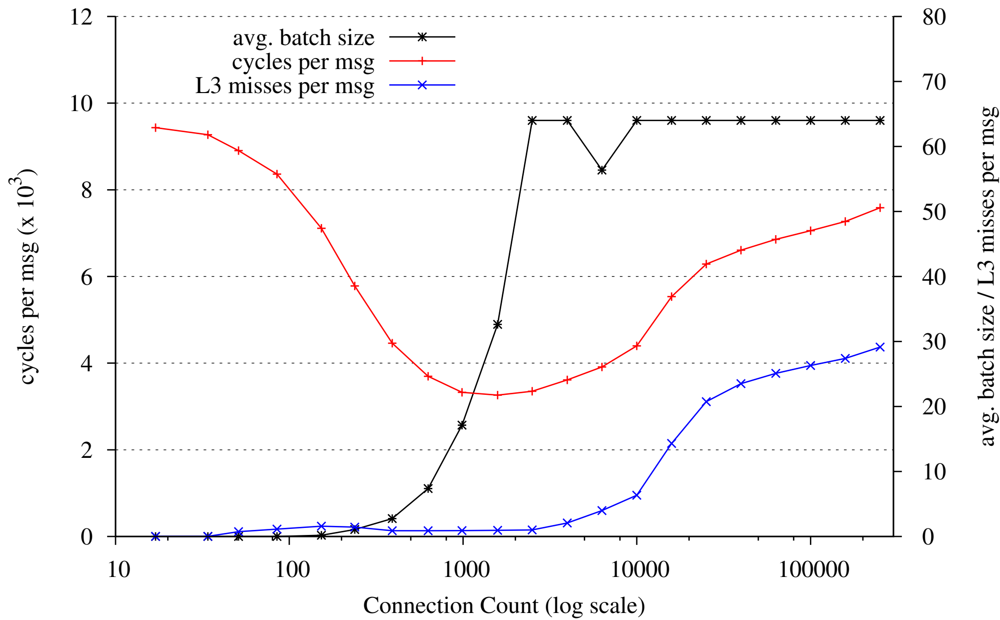


Figure 4 – Metrics for IX on the 4x10GbE configuration for varying established connections

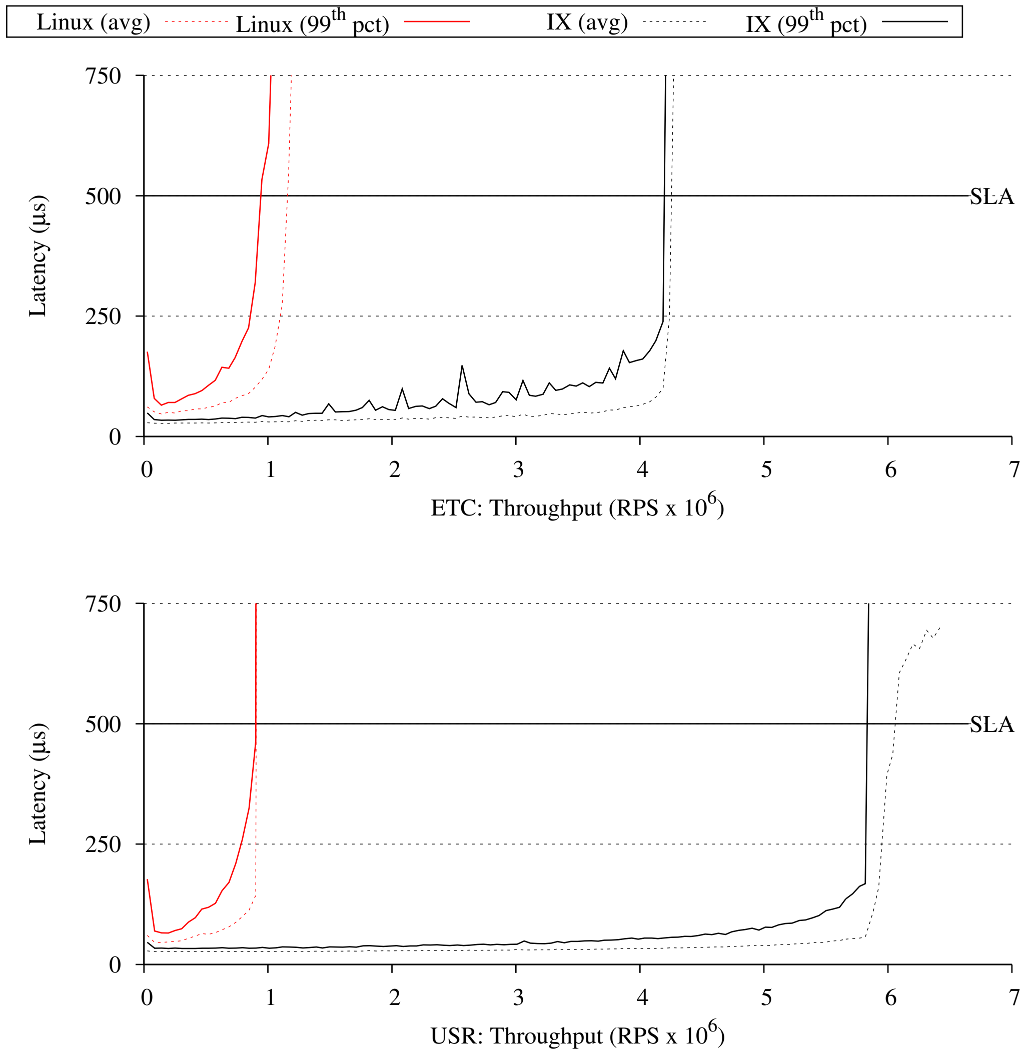


Figure 5 – Average and 99th percentile latency as a function of throughput for the ETC and USR memcached workloads.

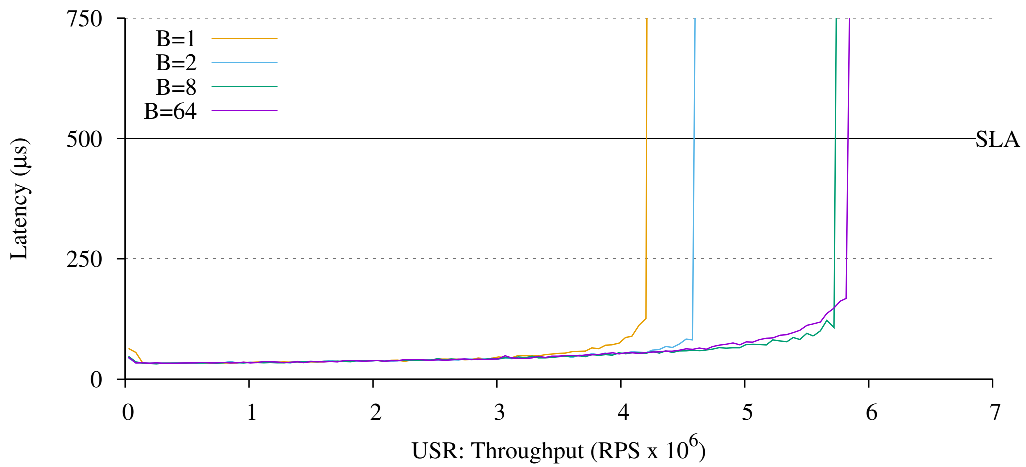
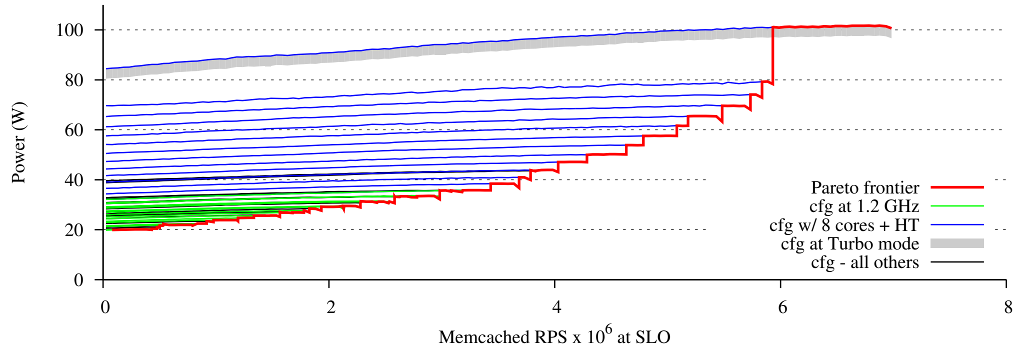
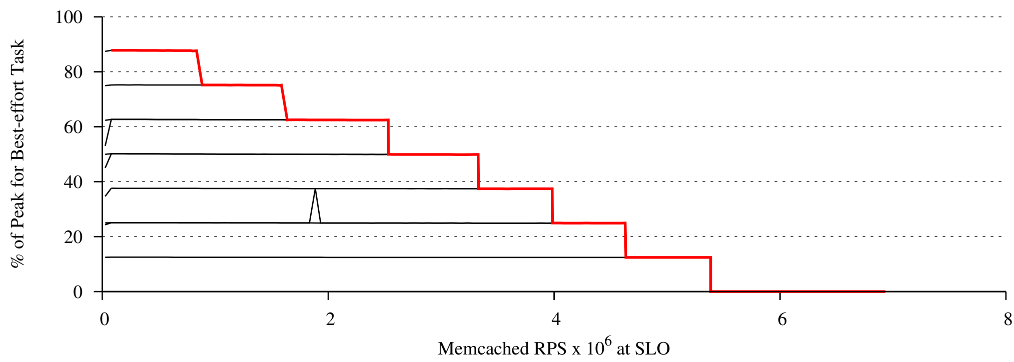


Figure 6 – 99th percentile latency as a function of throughput for `USR` workload from Fig. 5b, for different values of the batch bound B.



(a) Energy Proportionality



(b) Server Consolidation (IX)

Figure 7 – Pareto efficiency for energy proportionality and workload consolidation for IX. The Pareto efficiency is in red while the various static configurations are color-coded according to their distinctive characteristics.

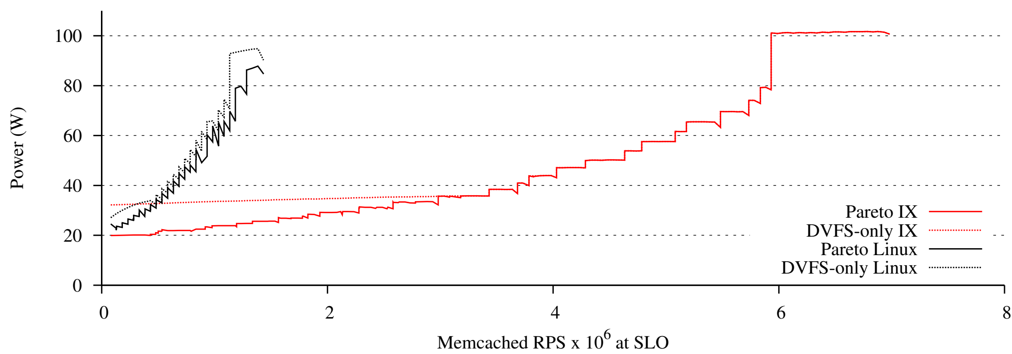


Figure 8 – Energy-proportionality comparison between the Pareto-optimal frontier considering only DVFS adjustments, and the full Pareto frontier considering core allocation, hyperthread allocations, and frequency.

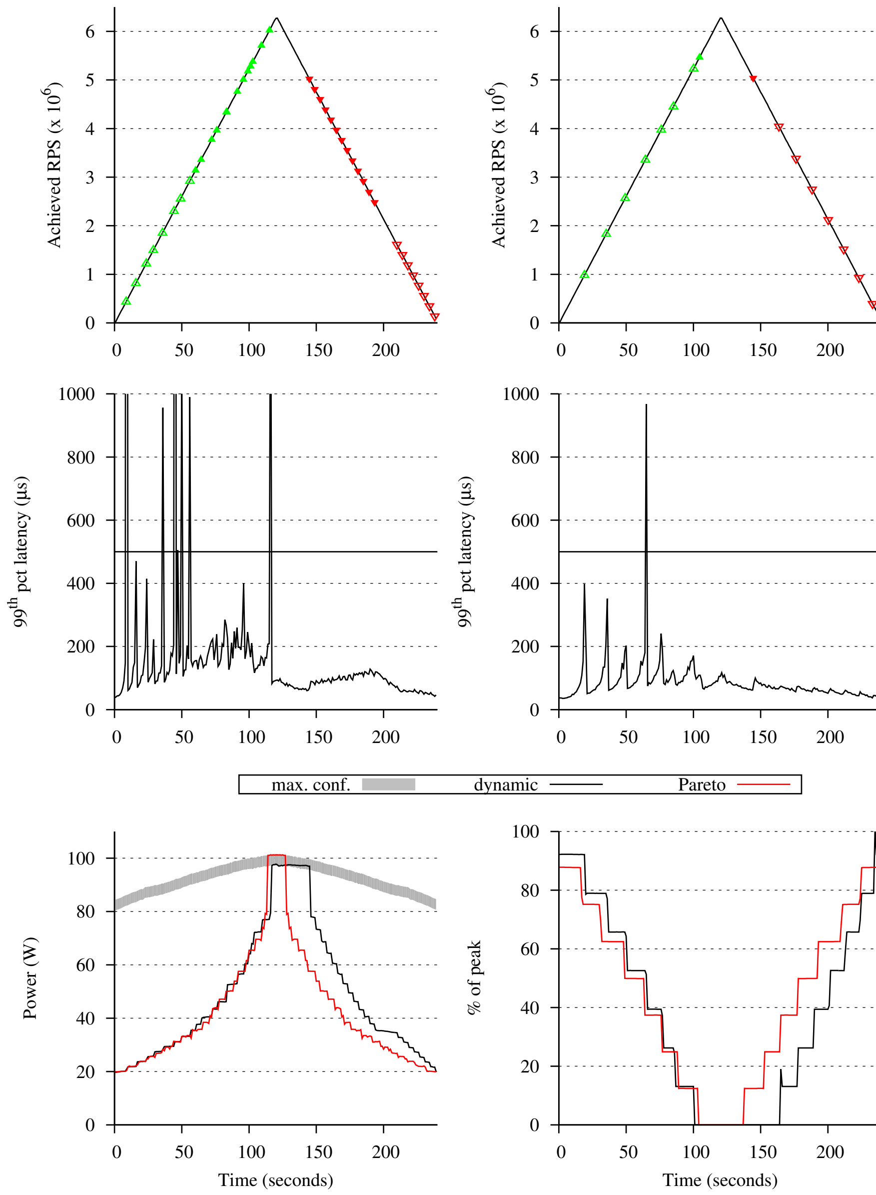


Figure 9 – Energy proportionality (left) and workload consolidation (right) for the *slope* pattern

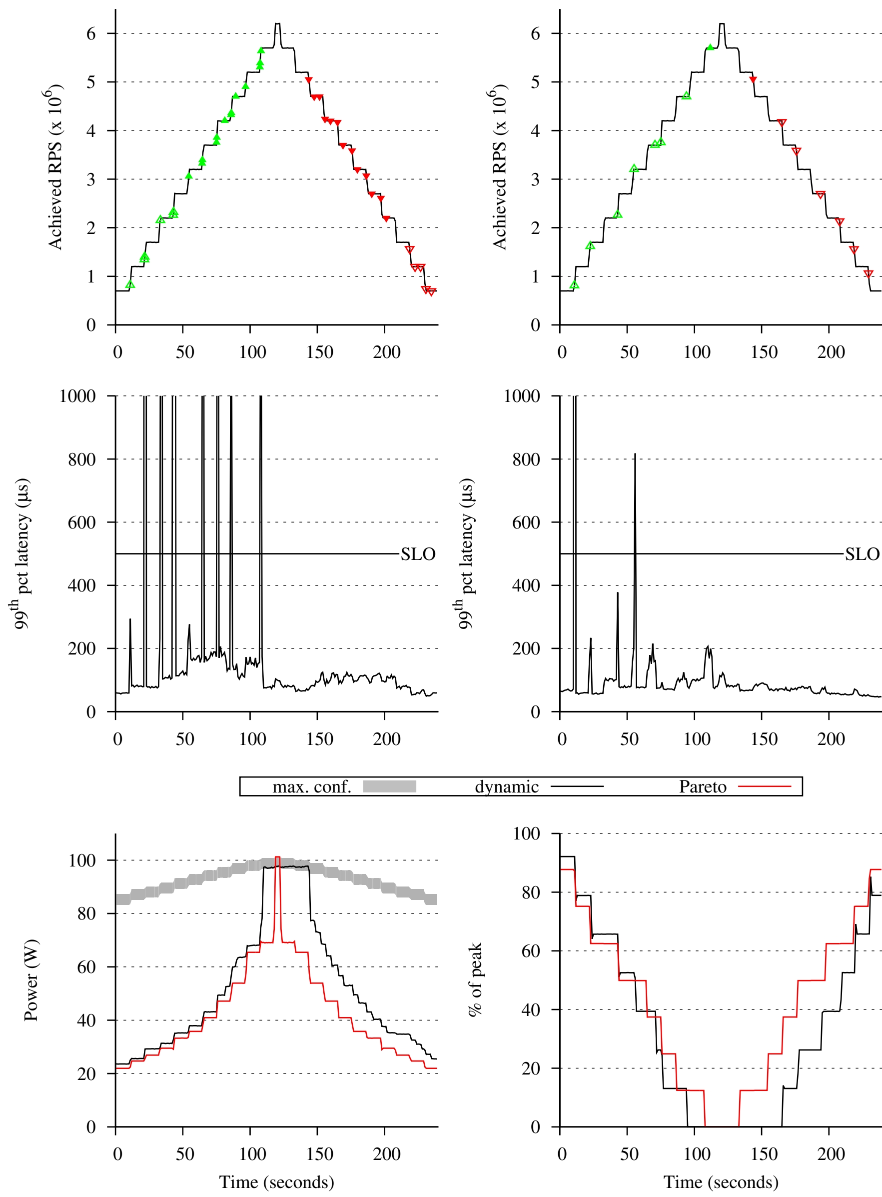


Figure 10 – Energy proportionality (left) and workload consolidation (right) for the *step* pattern

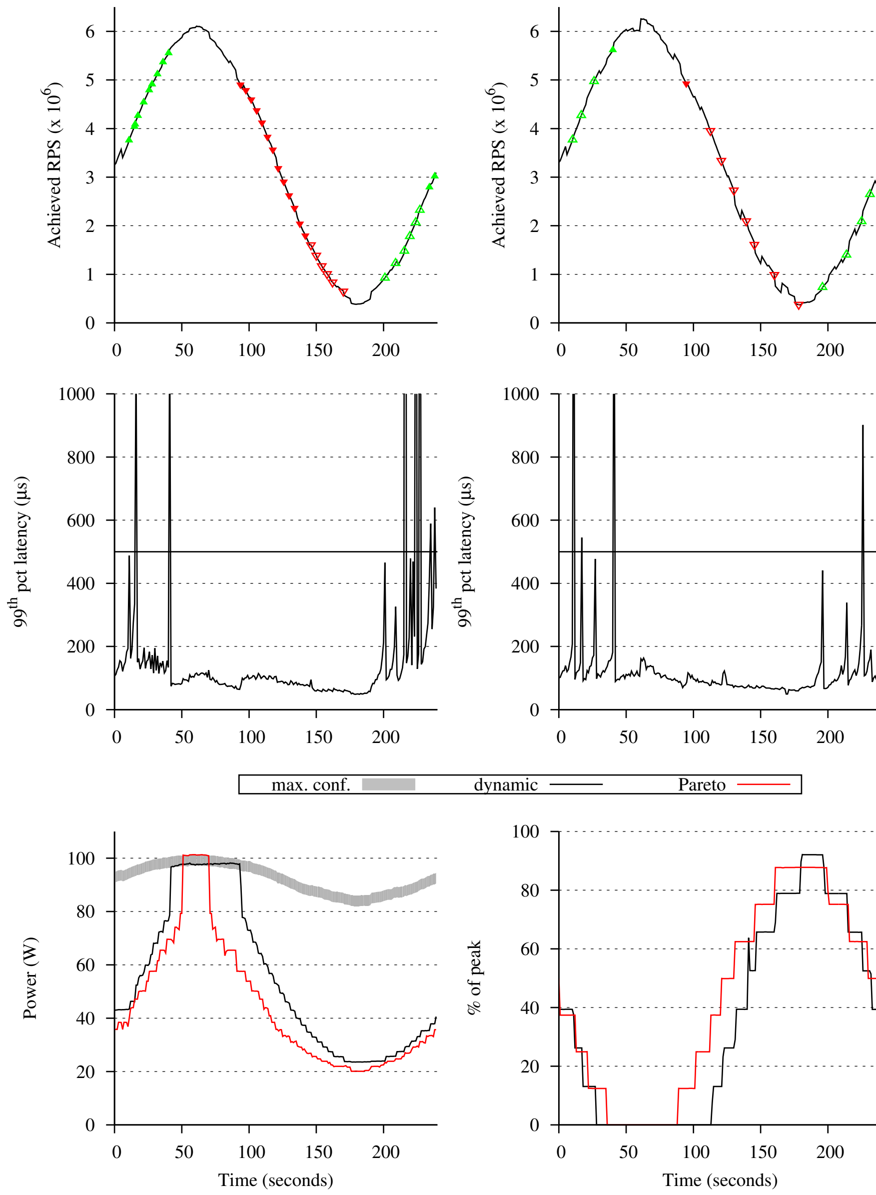


Figure 11 – Energy proportionality (left) and workload consolidation (right) for the *sin+noise* pattern