Data Science on FreeBSD/ARM64

BY MACIEJ CZEKAJ

Recently, ARM64 became a Tier I platform for FreeBSD. Since Semihalf has a long history of supporting FreeBSD on anything ARM, it was a logical step to use it in production. The test bed was unusual, however, since it was not yet another Web server or NFS storage array (which we have plenty of already), but a full-fledged Data Science lab.

The task at hand was to run a large-scale simulation experiment on the Marvell ThunderX2 ARM server. The simulation experiment resulted in the scientific publication and a chapter in the PhD thesis. The workload spans hundreds of CPU-hours for custom simulation software alongside the standard Open Source scientific toolkit, such as SciPy, Pandas, and Jupyter. The main bottleneck of the simulation system was RAM, while putting equal pressure on the disk I/O and data integrity. The software suite was originally developed for Linux and had to be ported to FreeBSD (by complying with POSIX).

ThunderX2 used in the experiment is a dual-socket 56-core ARM64 platform. The single CPU die has 28 cores in eight core complexes joined by the ring interconnect with shared L3 cache with cross-section bandwidth of more than 6TB/s. Each core may have up to 4 SMT threads totalling to 224 threads in the system. The 8-channel DDR4 interface for each die provides over 200GB/s of memory bandwidth for the whole system. The CPU dies are connected through CCPIv2 interconnect providing 600 Gb/s bandwidth. Looking at the specs, it seems to be the perfect target for memory-bound workloads.

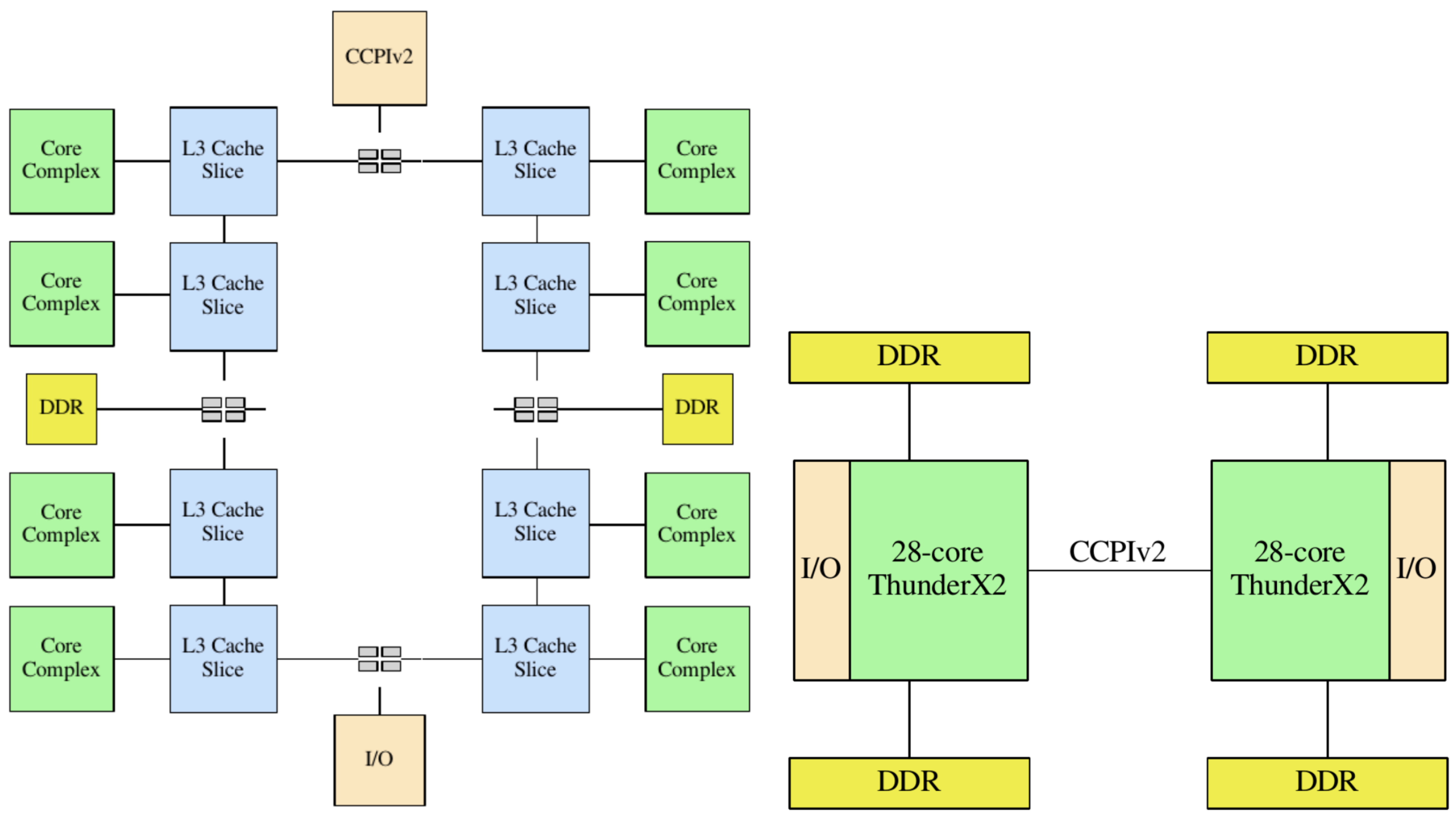

Fig1. The architecture of ThunderX2 system.

Originally used in the GNU Linux/x86 desktop environment, the simulation system had to be adapted to a parallel environment, possibly without too much programming effort. The central part of the system is a custom simulator software written in C++. The simulator accepts a recorded packet trace (PCAP stream) and produces a network flow database. The stream may come from a file or from another program (the mixer) which combines many packet streams together. The flow database is a custom binary format, which resembles the memory organization of a C table of structures. This format is both easy to serialize in C/C++ (by `frwrite()`) as well as easy to parse by Numpy (by `fromfile()`).

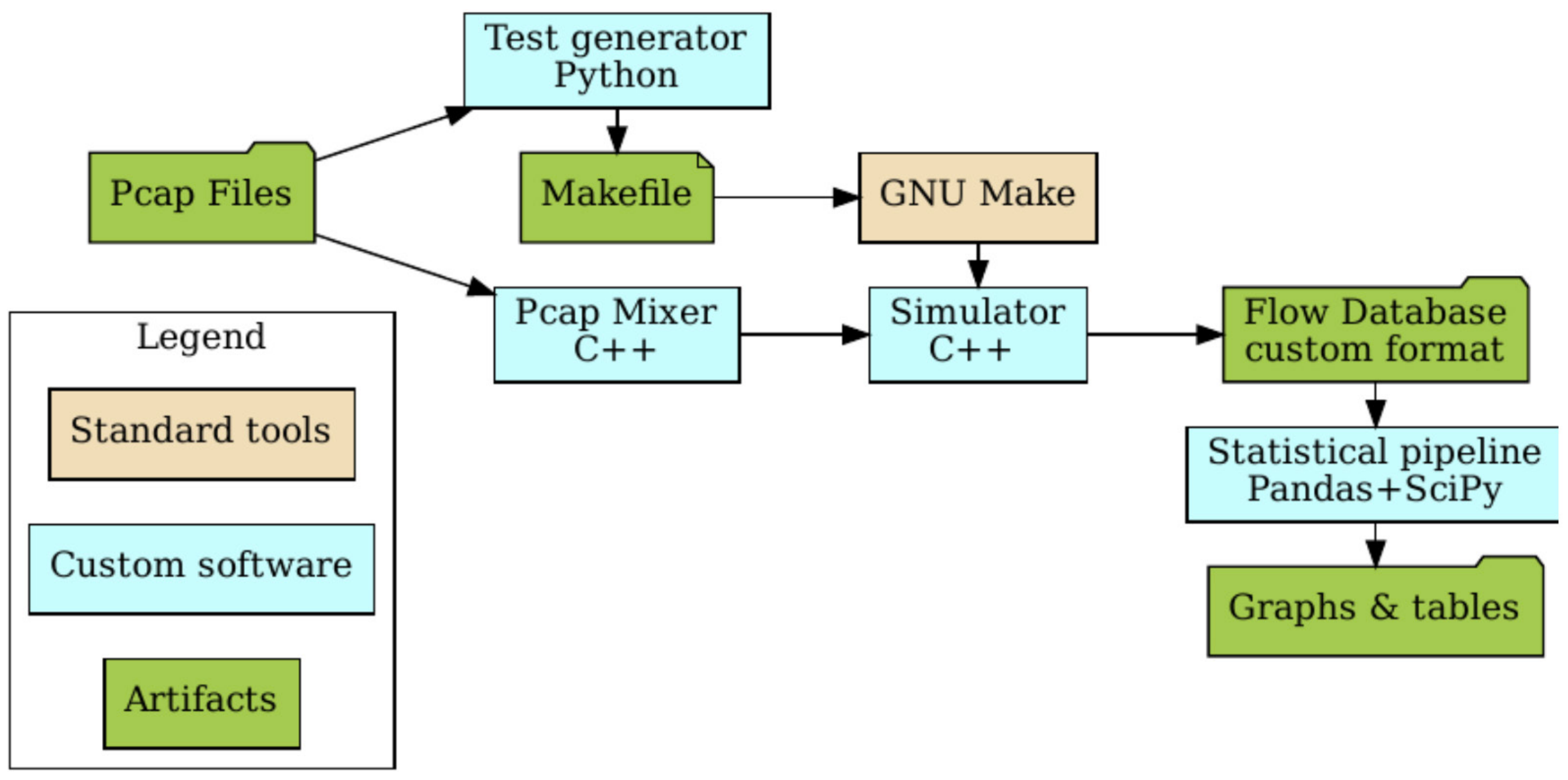

Fig2. The custom Data-Scientific pipeline executed on FreeBSD.

The next phase is controlled by statistical software in Jupyter Notebook. Each experiment produces millions of records occupying gigabytes of RAM, which mandates the use of an in-memory database for analytics in the form of Pandas DataFrame objects. The whole pipeline is described as a set of GNU Make job definitions.

The porting process from GNU/Linux to FreeBSD-12.2 was relatively simple. The C++ code base used mostly I/O system calls, which are part of the POSIX standard. Porting from GCC to Clang revealed a few issues about the code base itself, lending credibility to the common wisdom that using more than one compiler improves the code quality. The only functional issue was the usage of a hash function from the standard C++ library. The exact algorithm is implementation-dependent, so in order to keep the results reproducible, the hashing function source code must be provided. There were few performance issues with the C++ `iostream` library on FreeBSD. Granted, using text-based I/O was a design mistake in the first place, so the porting effort only amplified that inherent weakness. In summary, the porting of the C++ code proved to be the least concern and making it a multi-platform software improved the overall quality of the simulator.

The porting process from GNU/Linux to FreeBSD-12.2 was relatively simple.

Surprisingly, using the popular Python frameworks posed a bigger challenge than porting the C++ code. Popular scientific packages have many dependencies and usually are kept outside of a standard OS-specific Python stack. The essential challenge is to match the right version of Python, Numpy, SciPy, Pandas, Scikit-learn, and dozens of dependencies. In GNU/Linux the most popular way to resolve this conundrum is to use the binary Anaconda distribution. To my disappointment, the Anaconda dev team does not express any interest in supporting FreeBSD. The only alternative (apart from compiling everything from scratch) was to use Python Virtualenv. The fun started right away, when some of the packages were expecting GCC and others assumed Linux-specific include paths. After the painful process, all the essential packages were compiled. This should not be a surprise that Python packages heavily rely on third-party C or C++ libraries. Many Python packages are only language bindings to libraries written in C. Each time the package is installed, the third-party dependencies must be recompiled. It is worth keeping in mind that the whole Python stack is not totally independent of the base system. Thus, upgrading the FreeBSD poses a risk of repeating the whole process.

If the deployment of the Python stack was so cumbersome, was it worth it? Ultimately — yes — due to parallel computing. By default, the computation on DataFrame objects is single-threaded. However, the Pandarallel package provides a seamless parallelization through multiprocessing. Though not perfect, as it mandates copying the data, the speedup is still significant for CPU-intensive computations.

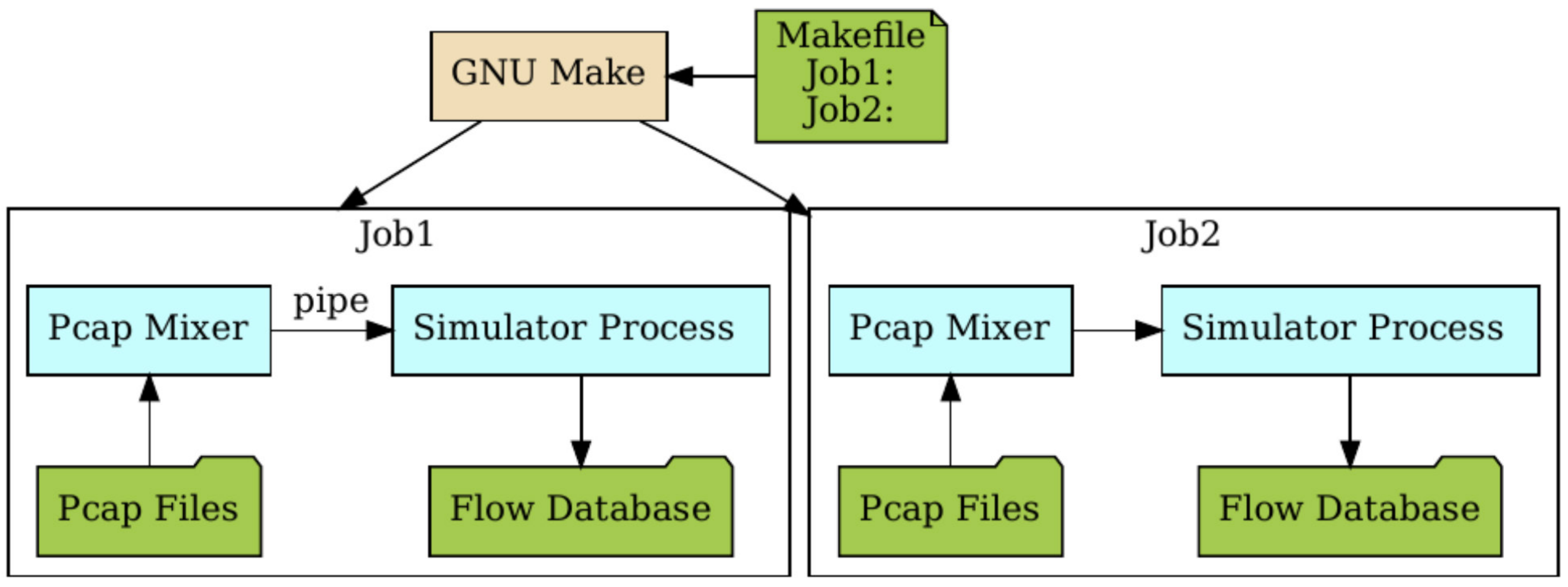

Fig 3.The parallelization scheme supervsed by Make.

The simulation system was designed to be single-threaded. The packet processing job must maintain an order of packets, so the central algorithm must remain sequential. The only viable means to scale the workload to many CPU cores was to exploit the coarse-grained parallelism in the workflow itself. The workflow definition contains more than 500 independent jobs. Each job lasts from several minutes to an hour, with the memory consumption from 10 to 30 GB (just for the data structures, so that was essentially an un-swappable resident set).

Luckily, the Make utility is capable of supervising a fixed number parallel jobs though the ubiquitous "-j" option. The challenging part was to match the varying memory requirements of the jobs with the number of processes. To my knowledge, there are no build systems that try to limit the number of jobs based on the memory pressure. They all consider only CPU load. Thanks to the infamous FeeBSD OOM killer and the decades-old UNIX wisdom accumulated in the Make utility, that turned out to be easier than expected. Each time when the number of jobs exceeded the RAM capacity, the OOM killer would pick the most memory-hungry process. This resulted in wasted CPU time, but kept the whole system stable and responsive. Also, the artifacts produced by the killed process were removed by Make, so data integrity was not compromised. This behavior is contingent on the correct job definition, since only explicitly defined Make targets are deleted.

If the deployment of the Python stack was so cumbersome, was it worth it?

The final version of the system was running continuously for over a week with intermittent supervision from the operator. The high demand for disk I/O bandwidth was met by the use of SSD drives backed by the ZFS filesystem. The overall stability of the platform was proven beyond any doubts.

The final conclusions for role of the FreeBSD/ARM64 as a scientific platform can be drawn in few points:

1. The platform provides excellent stability. The low overhead of the system and minimalist distribution leaves plenty of room for CPU-intensive or memory-intensive tasks.
2. The I/O subsystem can keep up with the most demanding workloads as long as the backing storage is solid state.
3. Porting the software from x86_64 to ARM64 architecture is mostly as easy as recompiling, provided that the developer follows the best practices of creating portable code. In case of porting from Linux from the same architecture, Linuxulator provides Linux ABI for native binaries.
4. Complex software stacks which are not supported by the system package manager can pose some challenges, if there are no alternative package managers. It is best to use shortcuts, with pre-built environments based on jails. As usual with the Open Source community, the software needs a critical mass of users to draw the attention of developers.

This yet another FreeBSD success story is a testimony to the effort of many developers which solidified the ARM64 port to the point where using the system is as ubiquitous as any x86 machine.

MACIEJ CZEKAJ is a lead s/w engineer at Semihalf, specializing in high-speed networking applications and drivers. He is a contributor to the DPDK project where he claims the authorship of one of the first ARM64 Ethernet device drivers (VNIC on ThunderX ARM64 server). He received his Comp. Sci. PhD program at AGH University (Kraków, Poland) on high-speed network acceleration.