OpenFlow in the Small

Raffaele Bolla
CNIT – ResearchUnit of the University of Genoa
DITEN – University of Genoa
Genoa, Italy

Roberto Bruschi
CNIT – ResearchUnit of the University of Genoa
Genoa, Italy

Chiara Lombardo
CNIT – ResearchUnit of the University of Genoa
DITEN – University of Genoa
Genoa, Italy

Fabio Podda
DITEN – University of Genoa
Genoa, Italy

Abstract— **Multi-core processors optimized for networking applications typically combine general-purpose cores with off-load engines to relieve processor cores of specialized packet processing tasks such as parsing, classification, and security. Unfortunately, modern embedded operating systems still lack of an effective and advanced support and hardware abstraction for optimally exploiting the above mentioned aspects. Starting from these considerations, this paper proposes a novel framework, "OpenFlow in the Small" (OFiS), specifically designed to provide a flexible hardware abstraction for a huge set of heterogeneous multi-core processors with advanced network off-load capabilities. The OFiS framework allows SW services/applications, or even only the operating system, activating and customizing off-load operations and their distribution to processor cores on a per-flow basis.**

Keywords— OpenFlow, Network Programmability, Network Processors.

I. INTRODUCTION

The evolution of network and networked device architectures (i.e., from routers to mobile phones and customer premises equipment) are two of the most important aspects of the Internet as we know it today, since they directly reflect the main open issues and the needs of current and upcoming network technologies and services. Network devices are becoming smarter and even more complex, in order to support new generation services/applications that often rely on a strong integration between network infrastructures and data-centers.

In the last years many ICT manufactures have evolved their device architectures (from routers, to mobile phones and customer premises equipment) by massively including sophisticated multi-core processors able to run embedded operating systems [1][2]. This choice was certainly driven by the need to avoid the increase of their internal clock or computational capacity, as suggested by Moore's law [3]. The resulting architectures consist of a complex mixture of flexible SW-based commodities and efficient HW-based off-loading functionalities.

Achieving high processing efficiency while providing full control over the traffic flow with minimal intervention from processor cores, however, calls for careful coordination among the various engines. In fact, on the contrary of non-programmable devices (e.g., based on ASICs) where performance levels are fixed and depending only on the HW design, the efficiency level of these platforms heavily relies on how effectively SW services exploit the underlying HW, and especially the parallelism between cores and HW off-loading functionalities.

Modern embedded operating systems (e.g., Linux) certainly guarantee a good abstraction of the underlying HW, and a friendly programming environment that might fasten new service development, as shown in [4], but most of the enhancements currently available on such OSs are not much effective when they have to cope with data received by high speed network interfaces. In fact, as already discussed in [5]-[8], when the cores receive or transmit packets from/to a same network interface, the performance level of the system dramatically decays.

Moreover, HW off-load APIs are often proprietary and, even if providing similar functionalities, may differ a lot according to the chip manufacturer. In order to improve the HW off-loading capabilities of multi-core systems, a standard communication layer would be needed. It should be flexible enough to provide some basic capabilities to a very wide set of devices, but also able to be extended and improved in presence of a more sophisticated hardware underneath.

Starting from these considerations, this paper proposes a novel framework, "OpenFlow in the Small" (OFiS), specifically designed to provide a flexible hardware abstraction for a huge set of heterogeneous multi-core processors with advanced network off-load capabilities. The OFiS framework allows SW services/applications, or even only the operating system, activating and customizing off-load operations and their distribution to processor cores on a per-flow basis.

This approach not only provides a simple high-level platform-independent interface to SW developers, but it also gives the chance of optimally exploiting the HW parallelisms, and of differentiating the operations on incoming traffic flows in order to better meet the application-specific requirements and the overall Quality of Service (QoS) level.

The name of the proposed framework is due to the core structure of its interface towards SW services and applications, which is structured in a similar way with respect to the OpenFlow protocol [9]: a template for identifying the packets of a flow, and a list of operations to be performed on all the packets matching the template. However, differently from OpenFlow, OFiS is devoted to manage traffic flows in a same complex chip (in the small).

The rest of the paper is organized as follows. Section II introduces some of the common issues in multi-core architectures, while the description of our framework can be found in Section III. Results are in Section IV, and conclusions in Section V.

II. Main Issues on Parallel Processors/Cores

From a general point of view, typical performance issues, which may sap parallelization gain, are raised when tasks on different cores share some data, or when the execution of a same task is migrated to another core. These lead to two well-known performance issues, namely data coherence and concurrency on shared data, which both introduce costly overhead in accessing and in processing shared data.

Data coherence issues arise from the hierarchical structures of memories of modern processors, which exhibit various levels of caches for quickly accessing frequently used data with low latencies. Depending on the processor, some of these levels are shared among cores, others unshared.

Shared data can be kept in only one processor's cache at a time, otherwise the core cache may drift out of synchronization (i.e., some cores may work on data that is not up-to-date). Consequently, whenever a core loads shared data to its local cache, all of the other processors caching it must invalidate their cache copies. Thus, this invalidation is very costly, since shared data can only be cached by one core at a time, but also, it forces the data to be loaded from the RAM or the higher level caches every time the processing core changes. This obviously introduces a non-negligible memory accessing overhead.

Both concurrency and data coherence are usually assisted by the operating system through serialization primitives (e.g., semaphores, locks, etc.) and suitable policies for task scheduling, respectively. If, on one hand, the above topics are well addressed by modern operating systems in a typical situation, they are still a big concern for HW/SW designers when tasks execution heavily relies on data received by high speed I/O hardware, and especially by network interfaces.

The idea of our framework starts exactly from these concerns: improving CPU off-loading while keeping a high flexibility level, so that our proposal would still confirm the versatility needed to apply to different architectures and, at the same time, follow future Internet evolutions. With this respect, our main purpose regards the introduction of mechanisms to optimize the internal processing of multi-core network processors, in particular to reduce the amount of complex operations that CPUs have to perform. A forwarding engine with such characteristics would combine the flexibility typical of open platforms to the off-loading needed to support more specific capabilities.

III. OpenFlow in the Small

The OFiS framework has been specifically designed to provide a flexible hardware abstraction for a huge set of heterogeneous multi-core processors with advance network off-load capabilities. This approach not only provides a simple high-level platform-independent interface to SW developers, but it also gives the chance to optimally exploit the HW parallelism, and to differentiate the operations on incoming traffic flows in order to better meet the application-specific requirements and the overall Quality of Service (QoS) level.

The main factor that makes OFiS particularly interesting is its ease of implementation: the framework described in the following is basic enough to deal with different HW architectures, but it can be easily extended in presence of more evolved HW enhancements to provide a more enriched implementation.

Moreover, as it will be extensively shown in Section IV, OFiS provides an extremely valid support to increase the performance level obtained through a fine-grained CPU specialization and off-loading.

A. *Framework Description*

The name OpenFlow in the Small wants to attest that the base principles are the same used in OpenFlow, but the main structure has been adapted to cope with systems with heterogeneous capabilities that can be available in different network/packet processors.

OFiS allows classifying incoming traffic into flows, and associating specific actions to them. In detail, incoming traffic is classified by means of a flow table. The flow table is composed by a number of entries $i = 1, \dots, I$ with $I \leq I^{max}$. The generic i-th entry is a triple, composed by:

- a flow descriptor f_i,
- a set of off-loading actions A_i^j with $j = 1, \dots, J_i$,
- a single redirection action R_i.

Note that if $I = 0$ the flow table is empty and if $J_i = 0$ there are no off-loading actions associated to the i-th flow.

Our flow descriptor f_i is a data structure coincident to the OpenFlow definition [10], which contains a pre-defined set of fields from packet headers at multiple layers (from layer 2 to 4). A variable-sized mask is associated to each descriptor field to provide a more aggregated flow representation.

The actions to be performed on classified packets are organized into two main typologies: off-loading actions A_i^j and redirection ones R_i. Both A_i^j and R_i are expressed by a numerical code specifying the type of operation to be performed, and a list of input parameters as needed. The type of action defines the number and the format of input parameters needed. When actions are stored in the flow table, the numerical code of the action type is followed by the parameters.

The presence of a single redirection action is mandatory on each entry, since it provides information on where classified packets have to be sent. Off-loading actions are optional, and one or more of them can be associated to the generic i-th flow.

As discussed in more detail in Section III.B, in our current implementation off-loading actions correspond to the modification of one or more fields on the packet headers, but they can also include the insertion of a pre-padding to provide some information in advance, for example on the protocol, to the central cores, hence reducing the amount of operations they have to perform.

B. *The Flow Actions*

A wide set of actions to be applied to the classified flows can be defined and implemented. However, their feasibility strictly depends on the underlying architecture where OFiS is applied. In fact, because of HW limitations, some capabilities may not be supported on different devices. For this reason, when the operating system tries to add a new action to the flow table, OFiS has to check whether it is supported and, otherwise, returns an error message.

As previously sketched, actions can be divided into two categories: redirection and off-loading. The first category includes discarding packets and packet redirection to a CPU or a subset of CPUs. The second one includes changing a field in the packet header, and can involve any header field at any level.

The following list provides an example of a simple, yet effective, set of actions that we have already made available in our prototype implementation:

- Redirection type 1: discard the packet;
- Redirection type 2: forward packet to a subset of CPUs;
- Redirection type 3: forward packet directly to a network interface;
- Off-loading type 1: in IP header, modify source IP, destination IP, ToS or TTL, respectively;
- Off-loading type 2: in TCP header, modify source or destination port, respectively;
- Off-loading type 3: in UDP header, modify source or destination port, respectively;
- Off-loading type 4: Add or remove a VLAN tag.

Many other action types can be obviously defined. In our implementation, the action choice has fallen on the manipulation of those fields that particularly suit our purposes: varying IP addresses and TCP/UDP ports, for example, we can perform NAT without Netfilter support, as will be shown in Section IV.C, hence remove some overhead from the CPUs. Redirection to a subset of CPUs guarantees the efficient exploitation of CPUs specialized in processing different traffic flows. Finally, it is important to remind that changing a field in the header implies the need for checksum recalculation, otherwise a packet could be considered invalid.

C. *The OFiS Fast-Path Procedure*

When a packet incomes, a number of fields, corresponding to the ones specified in the flow descriptor data structure, are extracted from its headers (the “parsing” operation with reference to Figure 1).

Then, the parsed fields must be used as a classification key to find the matching f_i descriptor, if any. If multiple matching descriptors are present, the first one is selected. If no descriptors are matching, a default redirection is applied (e.g., redirect to CPU 0). Upon a matching f_i descriptor selection, all off-loading actions A_i^j associated to f_i are sequentially applied to the packet. After all off-loading operations have been applied, the packet redirection R_i is finally performed.

D. *Managing Flow Entries*

The procedure to add or remove entries from the OFiS flow table is very simple, as the entire framework is driven by the local system (and not by a remote controller as in OpenFlow). In more detail, all the flow table modifications can be triggered by the SW running on the main processor.

For instance, the procedure used for adding a new flow entry is depicted in Figure 2. First, the module controls if there is enough room to add a new flow to the flow table. Then, all off-loading and redirection actions are examined to determine if they are supported by the underlying HW. If all these sanity checks succeed, the flow table is updated, and the engines start using the new configuration. Otherwise, the configuration request is aborted and an error message is reported to the user-space processes.

IV. PERFORMANCE EVALUATION

Test results reported in this section have been collected in order to evaluate the overall performance of OFiS with respect to a reference scenario in which no such enhancements are introduced. Our aim is to show how OFiS allows improving traffic classification and redirection, and communication between the HW and the operating system. These aspects are crucial for optimally exploiting parallelization in multi-core processors. Moreover, as described in Section III, OFiS has the clear advantage of easily adapting to any underlying architecture, while HW off-load APIs are often proprietary and, even if providing similar functionalities, may differ much according to the chip manufacturer.

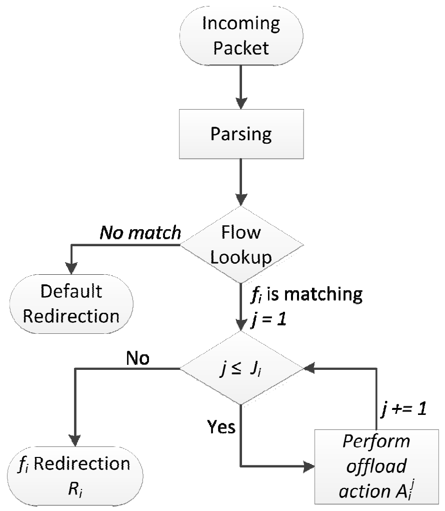

Figure 1. The OFiS fast path procedure.

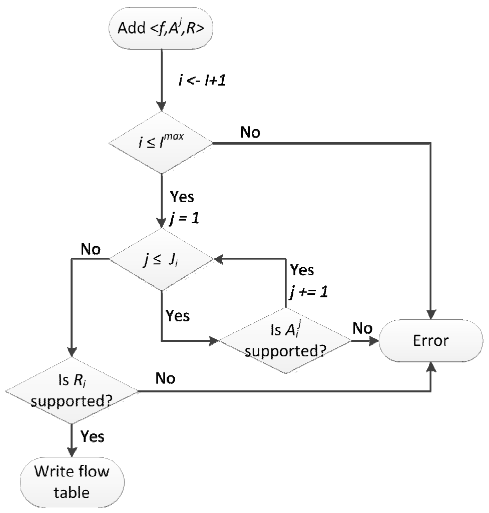

Figure 2. The procedure to add a new flow entry.

With these aspects in mind, we decided to focus our tests on the actions of redirection and off-loading. At this aim, we performed a first set of measurements on a scenario in which both IP and UDP traffic income in our network processor, to show how redirection can improve the overall performance. Then, in the second test, NAT is performed on incoming IP traffic using Netfilter and then OFiS, in order to compare the impact of the two methods on the system.

Both tests have been performed on the architecture presented in the next subsection. An Ixia Router Tester [11] has been used to generate traffic. The packet size is fixed to 64 Bytes, while offered loads are reported in each test.

A. Benchmarking Scenario

The reference HW architecture we adopted is the XLP832 [12]: a MIPS64-Release II processor with 8 EC4400 cores. Regarding the issues in parallel processors described in Section II, parallelization on the XLP is possible mainly because of the enhancements provided by the 2.6 Linux kernels, in particular thanks to the Symmetric Multi-Processor (SMP) support. For what concerns data coherency, CPU specialization obtained through the capacity of binding a process to a specific CPU (*CPU affinity*) can definitely reduce memory access, since the same information is likely to be exploited for multiple packets with the same characteristics (source, protocol, etc.), and can also reduce the number of CPUs trying to get the same data.

The main characteristic of the XLP technology is represented by the presence of specific hardware devoted to off-load operations usually managed by the CPU cores, called acceleration engines. Among them, the Network Acceleration Engine (NAE) is responsible for providing a huge part of the traffic management: in fact, incoming traffic is first processed by this engine, and is then directed to the CPUs only if further processing is needed, otherwise the CPUs are completely bypassed. Most of this process is performed by a set of micro-core engines, which are highly-programmable MIPS64 processors with access to a shared data RAM and a Content Addressable Memory (CAM).

In our implementation, the firmware of the micro-core engines has been developed in order to perform the operations in Figure 1, and to maintain a local version of the flow table. The flow table can be easily modified using a command-line application. To achieve maximum lookup speed, the CAM is used only for storing the flow descriptors, while the corresponding off-loading and redirection actions are in a local RAM.

For what concerns the management of the flow table entries, a Linux kernel module was designed to receive commands from user-space processes, to interpret them and to check their feasibility. If all these procedures succeed, the module finally applies the changes to the off-loading hardware by updating the flow table.

B. Redirection: UDP Test

In order to show the performance we can obtain by the OFiS adoption with respect to redirection actions, we represent a situation in which an application has to process specific traffic.

We consider a user-level UDP server that performs forwarding of UDP traffic at application layer. The traffic received by this UDP server has to be forwarded to another host. Of course, IP traffic is also processed and sent to the outer port.

In detail, in order to exploit OFiS potentials, we have bound the UDP server to a specific CPU and added a flow entry telling the system to forward UDP traffic to that CPU. To show the performance improvement obtained through this procedure, we compared it to a reference scenario in which these capabilities are not available.

Both scenarios have been tested forwarding the two traffic flows to the XLP. In detail, the system is receiving an IP flow incoming at 2.8 Mpkt/s, and a UDP one, whose speed varies between 14.880 and 74.404 Kpkt/s. Throughput and average latency of the UDP flow are measured at the varying packet rates. IP loss is also taken into account.

Figure 3 shows the results obtained on the two scenarios in terms of UDP traffic throughput. As we can see, in the reference scenario we experience some loss on UDP traffic even at the lowest rate. Such loss becomes particularly effective when the rate overcomes 59.523 Kpkt/s: throughput was already decaying linearly, but starting from this rate its deterioration becomes particularly visible and invalidating.

In the scenario with OFiS, instead, throughput does not decay or just loses a negligible percentage until 44.642 Kpkt/s. Although, for higher rates, loss becomes more visible, throughput decay is still lower than in the reference case.

The IP traffic flow transmitted during this test does not suffer any packet loss in presence of OFiS. In the reference scenario, instead, packet loss on IP traffic appears when the rate of UDP overcomes 59.523 Kpkt/s, and reaches 36% at 74.404 Kpkt/s. This result proves again the performance improvements obtained through redirection.

The same considerations are still valid considering latency results in Figure 4: we can see that, for all tested UDP rates, average latency is always lower when OFiS is introduced. This behavior is respected even in presence of loss on UDP traffic: as throughput falls at 44.642 Kpkt/s, latency starts to increase in the OFiS scenario but it still grows with a visibly slower trend with respect to the reference scenario.

These results confirm the effectiveness of OFiS to provide more than satisfying performance levels increasing what already obtained with the mere adoption of redirection.

C. *Off-Loading: NAT Test*

The following test is meant to expose the improvement in terms of off-loading obtained using OFiS. As previously mentioned, off-loading is absolutely a keystone towards a full exploitation of parallel processors. The introduction of our framework definitely represents a step forward in this direction.

With this respect, we decided to forward IP traffic and perform source NAT on all incoming packets. In the first test scenario, we used the most common method to manipulate traffic under Linux, which implies the use of iptables.

This well-known user-space application is the classic way to perform firewall operations. However, the filtering operations performed at kernel level by Netfilter are very onerous and can heavily increase CPU utilization. OFiS, instead, does not have such a heavy impact on the system overall performance, because it needs no intervention from the Linux kernel.

Results presented in the following show how the system benefits from OFiS introduction both in terms of network performance and global allocation of CPUs.

IP traffic has been transmitted to a subset of 8 CPUs at

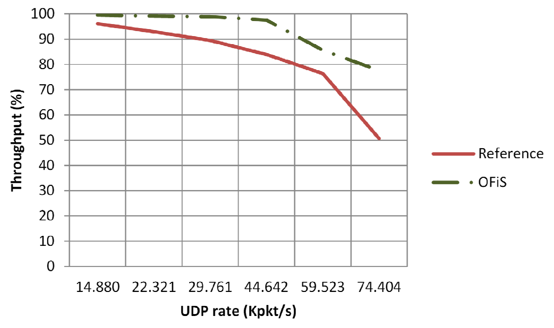

Figure 3. Throughput of the UDP traffic.

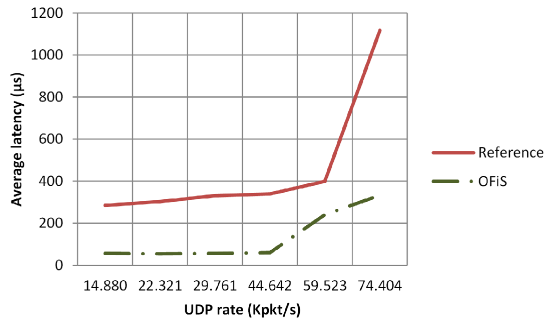

Figure 4. Average latency of the UDP traffic

various percentages of the maximum offered load, which, for 8 CPUs and in absence of operations on the incoming packets, has been measured to be 1.4 Mpkt/s.

Figures 5 and 6 represent the test results, obtained with iptables and OFiS. While the throughput obtained with OFiS reaches the offered load even at its maximum value, the scenario using iptables is affected by packet loss starting from 50% of the maximum load. As a consequence, the same load is characterized by an abrupt growth of latency, which instead is quite constant in the second scenario and presents lower values with respect to those obtained using iptables on the whole set of tested traffic loads.

A better understanding of these results can be obtained taking into account internal measurements performed on the XLP during packet processing. In detail, in order to provide a deeper analysis of the two scenarios, profiling has been performed on the CPUs involved in traffic forwarding. Using OProfile [13] at this purpose, we are able to effectively evaluate the CPU utilization of both each SW application and each single kernel function. Results reported in Figures 7 and 8 show the global time allocation of the CPUs processing traffic in case iptables or OFiS are used.

Considering Figure 7, representing the iptables scenario, the first aspect to be noticed is how scheduling and Netfilter are predominant among all events: throughout all offered loads, they account for more than 65% of the CPU allocation: we can state that Netfilter events are the main cause preventing the CPUs from going idle. For what concerns the other events, which all together allocate less than 40% of the CPU activity, we can see that they show a quite constant trend over the offered load. This is particularly odd considering packet header elaboration (pkth), which does not follow IP processing (ip).

Moving on to Figure 8, representing CPU utilization when OFiS is introduced, the absence of Netfilter is definitely the main difference with respect to the previous results. Scheduling is now visibly predominant throughout the offered loads, though its impact is stronger at lowest rates. Moreover, since throughput always equals the offered load, scheduling decreases as traffic rises, while traffic processing related events follow the traffic behavior at all percentages of the offered load.

V. Conclusions

In this paper, we introduced a framework to provide a flexible hardware abstraction for a huge set of heterogeneous multi-core processors with advanced network off-loading capabilities. Our framework, called Open Flow in the Small (OFiS), has been designed with the aim of optimally exploiting HW parallelism, and of differentiating the operations on incoming traffic flows in order to better meet the application-specific requirements and the overall Quality of Service (QoS) level.

It is common knowledge that concurrency on shared data and data coherence can heavily challenge any gain introduced by the adoption of multi-core architectures in network processors. The proposed framework is particularly suited to cope with these issues, since it provides an extremely valid support to increase the performance level obtained through redirection and off-loading actions. Among its advantages, it is worth mentioning flexibility: the framework is particularly easy to implement even on the most general purpose architectures, but it can also be extended without much effort in presence of more sophisticated HW.

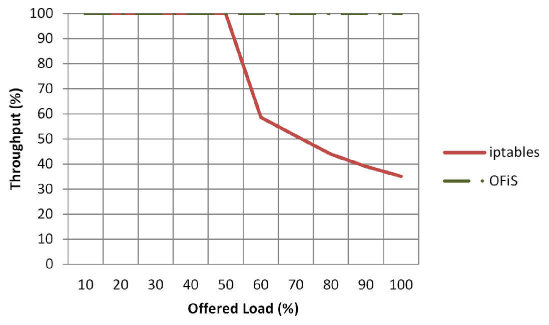

Figure 5. Throughput during NAT test.

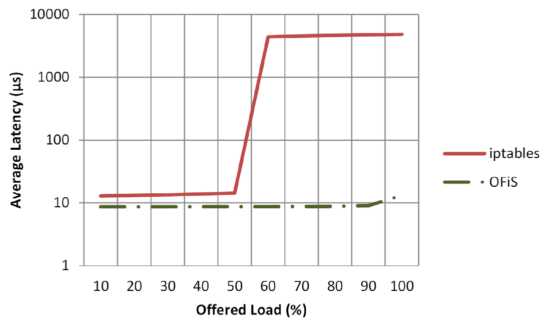

Figure 6. Average latency during NAT test.

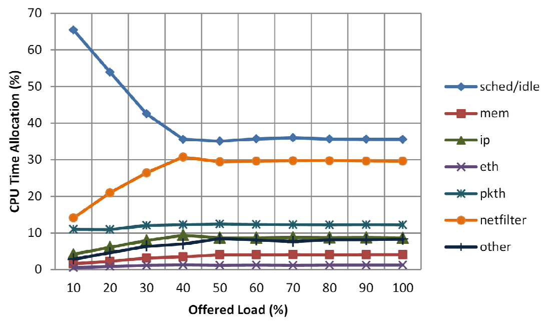

Figure 7. CPU utilization of 8 CPUs using iptables.

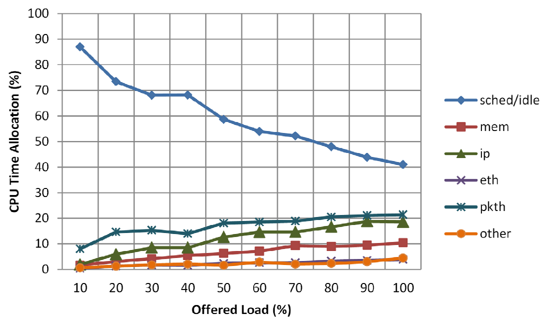

Figure 8. CPU utilization of 8 CPUs using OFiS.

In order to show the advantages obtained through the introduction of OFiS, we have presented an example of its implementation on the XLP processor. Test results stated that both network and system performance are visibly improved in presence of our framework.

ACKNOWLEDGMENT

This work was supported by the ECONET integrated project, funded by the European Commission under the 7th framework programme, theme ICT call 5, grant agreement no. 258454, and by the GreenNet project, funded by the Italian Ministry of University and Education under the program FIRB “Futuro in Ricerca”.

REFERENCES

[1] Vangal, S.; Singh, A.; Howard J.; Dighe, S.; Borkar, N.; Alvandpour, A. 2007. A 5.1GHz 0.34mm2 router for network-on-chip applications. Proc. of the IEEE Symp. on VLSI Circuits, June 2007, pp. 42–43.

[2] NetLogic MicroSystems. 2011. NetLogic Microsystems Unleashes Groundbreaking XLP® II, the World’s Most Powerful Multi-Core Communications Processors with Unparalleled Scalability to 640 NXCPUs™. Press Release, online, http://www.netlogicmicro.com/News/pr/2011/11-09-07xlpII.asp

[3] Moore, S. K. 2011. Multicore CPUs: Processor Proliferation. In “Top 11 Technologies of the Decade.” IEEE Spectrum, vol. 48, no. 1, pp. 27-43, Jan. 2011.

[4] Bolla, R. and Bruschi, R., 2007. Linux Software Router: Data Plane Optimization and Performance Evaluation. Journal of Networks (JNW) 2, 3, Academy Publisher, 6-11.

[5] Dobrescu, M.; Argyraki, K.; Iannaccone, G.; Manesh, M.; Ratnasamy S. 2010. Controlling parallelism in a multicore software router. Proc. of the ACM CoNext Workshop on Programmable Routers for Extensible Services of Tomorrow (PRESTO ‘10), Nov. 2010, Philadelphia, NJ, USA.

[6] Wu, Q.; Joy Mampilly, D.; Wolf, T. 2010. Distributed Runtime Load-Balancing for Software Routers on Homogeneous Many-Core Processors. Proc. of the ACM CoNext Workshop on Programmable Routers for Extensible Services of Tomorrow (PRESTO ‘10), Nov. 2010, Philadelphia, NJ, USA.

[7] Egi, N.; Greenhalgh, A.; Handley, M.; Hoerdt, M.; Huici, F.; Mathy, L.; Padimitriou, A. 2010. Forwarding Path Architectures for Multicore Software Routers. Proc. of the ACM CoNext Workshop on Programmable Routers for Extensible Services of Tomorrow (PRESTO ‘10), Nov. 2010, Philadelphia, NJ, USA.

[8] Bolla, R.; Bruschi, R. 2008. An Effective Forwarding Architecture for SMP Linux Routers. Proc. Of the 4th Int. Telecom Networking Workshop on QoS Multiservice IP Networks (QoS-IP 2008), Venice, Italy, 210-216.

[9] McKeown, N.; Anderson, T.; Balakrishnan, H.; Parulkar, G.; Peterson, L.; Rexford, J.; Shenker, S.; Turner, J. 2008. OpenFlow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69-74, March 2008.

[10] The OpenFlow Switch Specification, version 1.1.0, URL: http://www.openflow.org/documents/openflow-spec-v1.1.0.pdf.

[11] The Ixia XM2 router tester, URL: http://www.ixiacom.com/products/chassis/display?skey=ch_optixia_xm 2.

[12] The Netlogic XLP processor family, http://www.netlogicmicro.com/Products/MultiCore/index.asp.

[13] Oprofile, http://oprofile.sourceforge.net/news/.