Modular On-Chip Multiprocessor for Routing Applications

Saifeddine Berrayana, Etienne Faure, Daniela Genius, Frédéric Pétrot
{saifeddine.berrayana, etienne.faure, daniela.genius, frederic.petrot}@lip6.fr

Laboratoire LIP6, Département ASIM, Université Paris-VI, 4 place Jussieu, France

Abstract. Simulation platforms for network processing still have difficulties in finding a good compromise between speed and accuracy. This makes it difficult to identify the causes of performance bottlenecks: Are they caused by application, hardware architecture, or by a specificity of the operating system? We propose a simulation methodology for a multiprocessor network processing platform which contains sufficient detail to permit very precise simulation and performance evaluation while staying within reasonable limits of both specification and simulation time. As a case study, we show how a model can be developed for a IPv4 packet routing application, exhibiting the performance and scalability bottlenecks and can thus be used to reason about architectural alternatives.

1 Introduction

This paper proposes an efficient methodology for the performance tuning of networking applications on network processors by means of describing a flexible platform composed of hardware, software and operating system. Network processors are programmable routers used in specialized telecommunication equipment [1]. Among systems on chip they are distinguished by highly parallel hardware: one network processor can contain dozens or even hundreds of microprocessor cores, each executing a massively task parallel application.

In order to arrive at a reasonable performance, the application must be fine-tuned, and hardware and operating system must be adapted: Our approach is thus typically System-on-Chip. The remainder is structured as follows: First, we present our design methodology. We then show IPv4 as a case study and argue that our platform (hardware and operating system) can be seen as a shared-memory multiprocessor. Our validation methodology, as well as some performance results are shown before we conclude by outlining current limitations and perspectives.

2 Description methodology

The network processor modeling platform we took as a reference is ST Microelectronics' STepNP [2], which allows to study network processor architectures on transaction level [3]. Transaction level modeling reduces simulation time considerably, at the cost of a loss of precision. Our principal aim is to propose a methodology that enables to expose the multiple causes for multiprocessor-related bottlenecks, hence our need for a fine-grained simulation at register and signal level.

SystemC [4] is a hardware modeling language and simulation kernel that consists of a C++ class library. It permits the instantiation and assignment of models of hardware components and their signals in a hierarchic, building-block manner. Different levels of detail are possible, from coarse grain transaction level to fine grain register transfer level (RTL) [5].

We opted for a cycle-accurate simulation for which the values present at the connectors of a hardware component are known at every clock cycle. The SOCLIB component library [6] provides cycle-accurate, bit-accurate (CABA) simulation models, written in SystemC. Though equivalent to synthesizable models

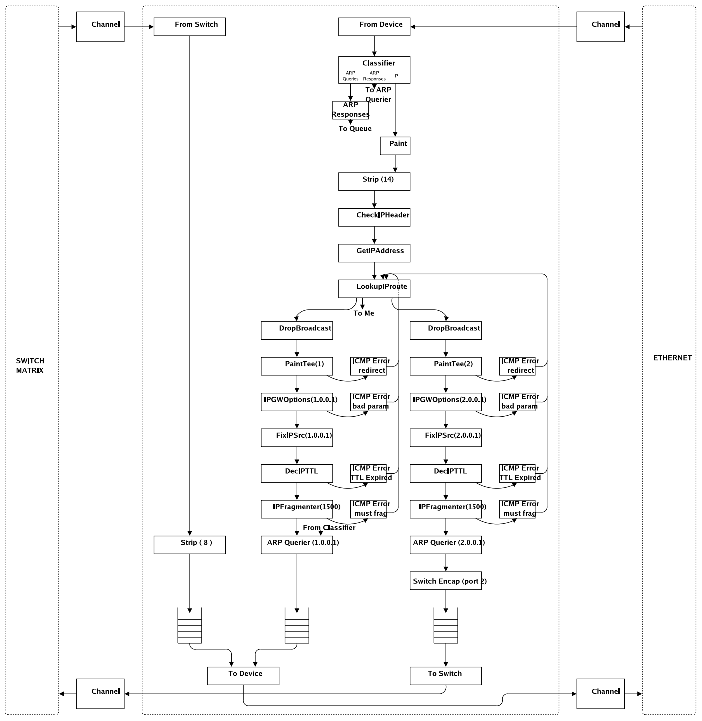

Fig. 1. Router Software Configuration

concerning external architecture, these models cannot be directly synthesized –for reasons of protection of intellectual property– and moreover, require much less processing power for their simulation.

The building blocks are easily connected by a unique VCI (*Virtual Component Interconnect*) interface [7], which permits to encapsulate arbitrary interconnection networks and to refer to them by a standardized protocol which fixes the number and type of signals required for a communication between components.

3 Porting an IPv4 Routing Application

A functioning system on chip consists of three parts: The software (the application itself), the hardware and the operating system. Together they build the platform; the challenge is to have them cooperate efficiently. In the next section, we will explain the choices we made to achieve this aim.

3.1 Software

IPv4 routing [8], well-known in the networking community, serves as our reference application. Essentially it consists in taking two or more input streams analyze their headers, and redirect the packets to the appropriate outputs. This application serves as an example in publications on Click [9, 10]. Click is a modular router configuration language consisting of two description levels: A simple high-level language to describe the structure of networking applications by composing so-called *elements* and a library of C++ components containing elementary functionality (such as IP packet header identification, buffering, discarding).

In section 3.3 we will restrict to discussing the application already modified to run on our platform: As can be seen in Figure 1, predefined elements are for example *FromDevice* and *ToDevice*, representing entering and exiting packet streams, respectively. Other important elements are *Paint* and *Strip* to mark packets for loops and delete a packet header. Within compound elements, several other elements' functionalities are summed up hierarchically. The central functionality of IPv4 lies in such a compound element, *LookupIProute*, which determines the route taken by an individual packet according to the information in its header. The packets pursue their course along two possible routes: In the direction of *ToDevice* if the packet is damaged, in the direction of *ToSwitch* otherwise. A third exit, *ToMe*, sends ARP requests/responses to the router itself.

Let us now explain the modifications we made for SoC implementation. The first modification concerns I/O. In order to have the application running on one element of a (multiprocessor) network processing platform (one Network Processing Unit, NPU), we have to take into account the fact that its interfaces are asymmetrical (see figure 2). One is the Ethernet link and as such is already treated correctly by Click; the other however is the junction with an on-chip switching matrix. This matrix also uses headers for routing, which have to be constructed and grafted at packet emission and deleted when the packet arrives, respectively.

A consequence of this modification is the addition of these two modules *SwitchEncap* and *FromSwitch* which do not appear in the initial list of Click modules. They serve to encapsulate an Ethernet sequence in the header used for internal routing in our chip, and to retrieve an Ethernet sequence after this internal routing, respectively. Moreover we wish to limit complexity. The second modification is due to the structure of the router we use. Our network processor only has two network interfaces connected to the interconnection matrix; likewise IPv4 only has four communication points: Two each for ingress and egress packets.

Thus, it is known that the packets arriving from this interface have already been treated by another block, and are moreover destined for the Ethernet link to the outer world. We can conclude that the verification and routing have already been accomplished by another unit. The only work still to be done is to direct the block to the egress interface. For this reason, Figure 1 shows an additional straight line between *fromSwitch* and *toEth*. The only work remaining is to strip the header (delete the first 8 bytes) for the interconnection matrix routing which is expressed in element *Strip(8)*. The modified application is shown in Figure 1.

3.2 Hardware Architecture Model

The application has been made as independent as possible from the context in which it is used, retaining the essentials of its structure. We look for a modular system architecture where we can easily add, exchange, and regroup hardware models. Such a system is shown in Figure 2. The first step in a modular approach is to define an elementary (SystemC) building block which can be instantiated to the required quantity. This building block is itself a router, and elementary in the sense that it has only two interfaces. This choice enables us to treat platforms with only two interfaces as well as to dimension our platform according to the number of network interfaces required. We have thus a two-level hierarchy of one router composed of four NPUs, interconnected by a switch matrix, each with a certain number of processors inside. From the hardware architecture point of view, two separate parts are to be realized on-chip (we will concentrate on the latter):

- an interconnection matrix that links network processors; possible simple but realistic topologies are a full crossbar for a small number of processors, a fat tree for a larger number [11]
- the network processor itself, i e. the unit which does the packet treatment

The hardware architecture we implement was originally inspired by the architecture developed by ST Microelectronics in the context of the STepNP platform. It

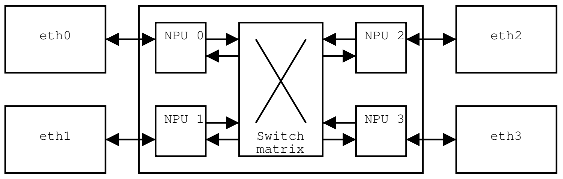

Fig. 2. Router with four network interfaces

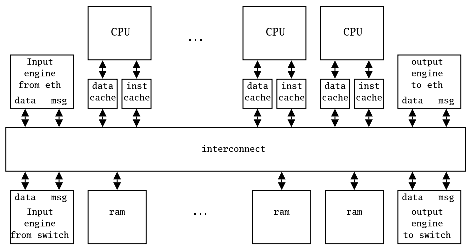

Fig. 3. Network processing unit hardware architecture

regroups multiple processors, two network interfaces around a on-chip interconnection network. The platform contains on the one hand calculation units, represented by MIPS R3000 processors with their caches, as well as SRAM memory containing the application code and data; all of these already exist as SOCLIB CABA models. On the other hand, we designed dedicated I/O co-processors to insert Ethernet packets into the system and to extract them, once they are treated. There are two such co-processors for each network interface, one each for ingress and egress, named respectively *input-engine* and *ouput-engine*. The architecture is shown in Figure 3. All these elements communicate via an on-chip interconnect carrying a VCI interface. Note that the interconnection network used in the simulations is a virtual model called VGMN (for VCI Generic Micro Network) whose parameters are the number of ports and latency. We are well aware that some precision is lost by using this abstract model instead of a real interconnect model.

A first noteworthy advantage of modularity is the fact that the number of co-processors is fixed for each NPU, whatever the number of interfaces that have to be connected to the system. In particular, this avoids to take into account problems due to saturation of interconnect bandwidth in the presence of multiple packet injectors. It will consequently be much easier to extrapolate our performance results to a larger model composed of several instances of the basic model and functioning independently of each other.

3.3 The Embedded Software

Once the functionality of our application being fixed, the next step is to port it to the parallel target architecture. Basically, there are two options available: On the one hand, exploiting coarse grain parallelism, which means looking for independent treatments and transform them into separate communicating tasks. On the other hand, duplicating the application in order to obtain identical clones, each treating an IP packet throughout its passage.

The simple kind of routing application we used as benchmark has relatively weak intrinsic parallelism, as one packet is treated by a sequence of functionality corresponding to Click elements. In [12], the authors propose to decompose one instance IPv4 into threads, with the Fifos as cutting points. As existing routers treat hundreds or thousands of structurally identical tasks in parallel from end to end, we consider it more promising to have all tasks execute the same code on different packets. One RAM contains the global variables, while all other RAMs are allocated to one processor each. However, the accesses to the shared memory are critical, as will show the experimental results.

3.4 Operating System Mutek

Once application and hardware fixed, it remains to determine the operating system. Our choice was the Mutek[13] micro kernel. This micro kernel is able to handle multiple tasks running on multiple processors. It provides a C standard library as well as support for POSIX threads.

An important point is that we wish to assign tasks statically to the processors, in order to avoid the cost of task migration between processors, and to avoid migration-related coherence problems. Mutek gives us that possibility.

Once the decisions on kernel and scheduling are made, it remains to determine the number of tasks assigned to on single processing unit. We made performance measures concluding that the time required for context switching outweights time gained by data latency. In the experiments presented here we show one task statically assigned to one processing unit.

4 Validation and Performance

The hardware and software part of our network processor have to be tested together. The entire platform simulated under SystemC will serve as proof of concept. The validation is made using several Ethernet packets benchmarks injected using the *input engine* and analyzed by the *output engine*.

The most relevant performance measure is the throughput in bits per cycle our processor can achieve. Our system is globally synchronous, i e. all of its components share the same clock domain. Thus, we can choose a clock cycle as basic time unit. On the other hand, we have left open the choice of the frequency; clearly, the higher the frequency at which our system works, the better its performance. Our first measure will serve as baseline throughout our experiments: The throughput the system guarantees when one single processor is instantiated, which is the maximal throughput of a single NPU.

When adding processors to our NPU, as can be expected, the performance is not linearly improved. The more adequate question is, how many processors can be added in order to still obtain a profit? Running the simulation during 2 000 000 cycles, and counting the number of 32 bit words arriving at each egress interface allows to neglect the impact of boot time (around 16 000 cycles); also packet(s) that have been treated but not yet been emitted are ignored. All IP packets have 56 bytes, which constitutes the worst case in our application context. This simulation yields in total 689 words of 32 bits each that have been read at egress, a throughput of 0.011 bit/cycle. This rather bad result is due to the non-optimized application. The weak throughput actually corresponds to an average time for treatment of 40 000 cycles for an IP packet. However, keep in mind that our goal is to exhibit performance bottlenecks on a detailed level, for a variety of architectures. Details on the execution times of the individual functions are summed up in table 4 for the case of a packet entering via the Ethernet link and exiting via the interconnection matrix. This corresponds to the longest possible path which a packet can take between two interfaces (around 110 000 cycles).

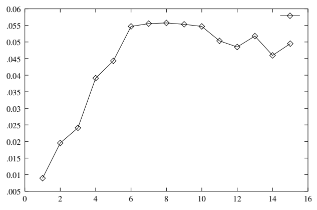

Fig. 4. Number of processors and resulting throughput in bits/cycle

This is nevertheless our reference for multiprocessor experiments. The first consists in simply varying the number of on-chip processing units. Figure 4 illustrates our results, presenting on the x-axis the number of MIPS R3000 processors, on the y-axis the throughput achieved, in bit/cycle. The measurements show that the best throughput is reached for 8 processors, but it is only slightly higher than that for six processors. All units added after the sixth will not yield a significant improvement, they rather lead to performance degradation. The reason for this is that all processors share the same resources, particularly the memory banks where the IP packets are copied to before or after being treated. This contention is only due to the fact that there is only one memory bank for all tasks, not because the data are shared. Thus, we distribute the memory around the interconnection network by replacing the four big memory segments used by the ingress/egress co-processors by as many segments as processors. Such a segment has to fulfill three functions: 1) contain the local data of the thread, 2) allow the ingress processors to write the entering packets there, 3) supply a space for the processor to copy the outgoing packets.

More precisely, each memory bank will be divided into three non overlapping regions. The second region is subdivided into two sections, one for each *input engine*. As none of the two co-processors knows the behavior of the other, we avoid to make them share variables or address space to not further complicate matters.

This system architecture modification permits a significant performance gain, also for larger numbers of processors. The graph for this second experiment is shown in Figure 5. Here, the performance peak is reached at 27 processors, with a throughput of 0.3 bit/cycle. Here again, a raise in performance to a certain point is followed by an abrupt degradation, caused by contention for the access to shared resources, more precisely, the access to the two segments of shared memory which have remained unchanged and are used by all threads: The code segment and the segment containing the global variables of the system. The presence of contention means that several initiators wish to establish a communication with the target at one time: In consequence, at least one initiator stays blocked waiting for access to the the resource.

The graph in Figure 6 shows the number of cycles due to bottlenecks stemming from accesses to global data in the RAM during an interval of time of 2 000 000 cycles. A closer look reveals that for 28 and more processors, the number of conflict cycles exceeds 1.8 million, which means that the system is paralyzed. Taking a closer look, most of the conflicts happen when processors try to access the RAM containing the operating system data. At the same time, they stay in the scheduler function, which wakes up another thread ready to run when the current thread falls asleep. In our case, there is only one thread per processor, thus we decided to prevent this function from being executed. To do this, the thread should never stop, even if it cannot access the ressource. We changed the lock associated with the shared ressources (input and output engines) from a mutex to a spinlock. From the thread point of view, both have the same behavior: A function call will return only when the resource is available. From the OS point of view, a mutex will suspend the calling thread if the resource is unavailable, whereas a spin lock will continue to demand the resource. More information about mutex and spin locks can be found in [14].

New performance results after this software modification are shown in figure 8: The throughput is still growing for more than 27 CPUs, the maximal throughput of 0.56 bit/cycle is reached with 45 processors. Figure 9 compares the three different sets of results we obtained with the different implementations. Obviously, the last change, replacing mutex with spinlocks, does not improve greatly unless we use more than 28 CPUs. This change does not or only marginally improve the throughput per processor, but it allows far more processors to share the same resources without any loss of performances.

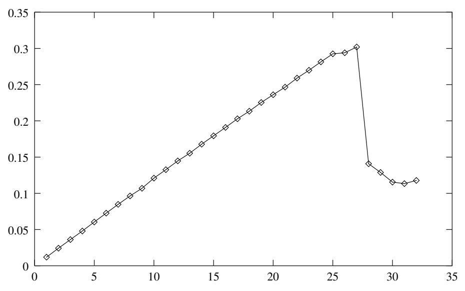

Fig. 5. Throughput in bits/cycle for the distributed memory implementation as a function of the number of CPUs

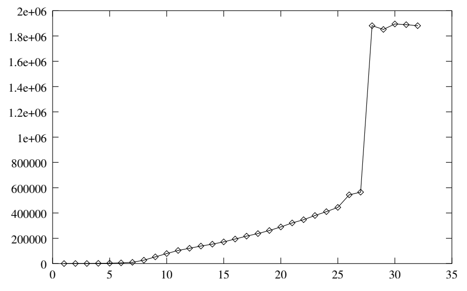

Fig. 6. Number of conflict cycles for the data RAM as a function of the number of CPUs, for a period of 2 000 000 cycles.

Element	execution time (in cycles)	ratio	Element	execution time (in cycles)	ratio
Acquisition	30534	27.14	PaintTee	346	0.31
Classifier	608	0.54	IPGWOptions	634	0.56
Paint	25332	22.52	FixIPsrc	314	0.28
Strip	356	0.316	DecIPTTL	998	0.89
Checkipheader	8506	7.56	IPFragmenter	422	0.38
Getipadress	2934	2.61	EtherEncap	15308	13.61
Lookupiprouter	598	0.53	Extraction	25218	22.42
Dropbroadcast	394	0.35	Total	112502	100%

Fig. 7. Execution time in cycles required for each element (single-thread, MTU = 56)

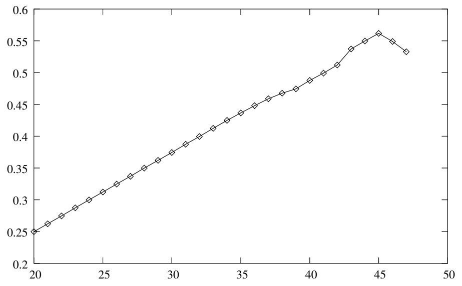

Fig. 8. Spin lock memory implementation: Throughput in bits/cycle, relative to the number of processors

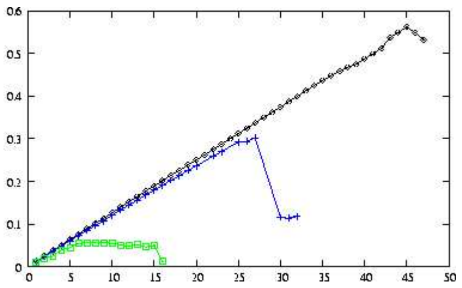

Fig. 9. Three sets of results obtained for the three different architectures, for the same simulation time.

5 Conclusion and Perspectives

By specifying and implementing a simulation model for a network processing platform, we have fulfilled two goals. Firstly, we have determined a detailed and efficient simulation system. The cycle accurate abstraction level is a good compromise between precision and simulation speed. The simulation speed allows us to consider real applications, while the level of detail allows to observe in detail the behavior of our system and helps remedy its insufficiencies. The second goal concerns architecture itself: We have successfully described a network processor core as well as run a multi task application. If performances remain modest, this is mainly due to the software part which will have to undergo profound optimization. Results are on the other hand only slightly impaired by the multiple instantiation of our processing unit.

Future work will take several directions: Our next step will be to sum up classes of networking applications by a few "generic" application templates that reflect the typical thread structure - examples for such templates are Quality of Service or Classification applications. Secondly, the use of a more realistic network-on-chip model would improve the precision of our results concerning contention. Thirdly, parts of the multi-threaded micro kernel could be implemented in hardware. Finally, our performance studies and the actual implementation have shown the urgent need for debug tools to determine critical resources and time each processor takes for each function. This raises the need for debugging tools for SystemC.

Our platform clearly constitutes a starting point for further experimentation; it is very open in the sense that it allows for a large range of applications from IPv4 routing to traffic analysis and encryption protocols.

References

1. N. Shah: Understanding network processors. Master's thesis, Dept. of Electrical Eng. and Computer Science, Univ. of California, Berkeley (2001)
2. Paulin, P., Pilkington, C., Bensoudane, E.: STepNP Platform. Ottawa, Canada (2002)
3. L. Cai and D. Gajski: Transaction level modelling in system level design. Tr, Univ. of California, Irvine (2003)
4. Open SystemC Initiative: SystemC. Technical report, OSCI (2003) http://www.systemc.org.
5. Groetker, T., Liao, S., Martin, G., Swain, S.: System Design in SystemC. Kluwer (2002)
6. SOCLIB Consortium: Projet SOCLIB: Plate-forme de modélisation et de simulation de systèmes integrés sur puce (the SOCLIB project: An integrated system-on-chip modelling and simulation platform). Technical report, CNRS (2003) http://soclib.lip6.fr.
7. VSI Alliance: Virtual Component Interface Standard (OCB 2 2.0). Technical report, VSI Alliance (2000) URL=http://www.vsi.org/library/specs/summary.htm#ocb.
8. Baker, F.: Requirements for ip version 4 router. Internet Eng. Task Force, ftp://ftp.ietf.org/rfc/rfc1812.txt (1995)
9. E. Kohler: Click system free software. URL=www.pdos.lcs.mit.edu/click (1995)
10. Kohler, E.: The Click modular router. PhD thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science (2000)
11. Andriahantenaina, A., Charléry, H., Greiner, A., Mortiez, L., Zeferino, C.: SPIN: a scalable, packet switched, on-chip micro-network. In: Design Automation and Test in Europe Conference (DATE'2003) Embedded Software Forum, Muenchen, Germany (2003) pp. 70–73
12. Chen, B., Morris, R.: Flexible control of parallelism in a multiprocessor PC router. In: Proceedings of the 2001 USENIX Annual Technical Conference (USENIX-01), Berkeley, CA, The USENIX Association (2001) 333–346
13. Pétrot, F., Gomez, P., Hommais, D.: Lightweight implementation of the POSIX threads API for an on-chip mips multiprocessor with VCI interconnect. In Jerraya, A.A., Yoo, S., Verkest, D., Wehn, N., eds.: Embedded Software for SoC. Kluwer Academic Publisher (2003) 25–38
14. Tanenbaum, A. In: Distributed Operating Systems. Prentice Hall (1995) 169–185