POLITECNICO DI TORINO
Repository ISTITUZIONALE

Creating portable and efficient packet processing applications

Original
Creating portable and efficient packet processing applications / Morandi, O., Risso, F.G.O., Rolando, P., Valenti, S., Veglia, P.. - In: DESIGN AUTOMATION FOR EMBEDDED SYSTEMS. - ISSN 0929-5585. - STAMPA. - 15:1(2011), pp. 51-85. [10.1007/s10617-011-9072-8]

Availability:
This version is available at: 11583/2381917 since:

Publisher:
Springer

Published
DOI:10.1007/s10617-011-9072-8

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright
Springer postprint/Author's Accepted Manuscript

This version of the article has been accepted for publication, after peer review (when applicable) and is subject to Springer Nature's AM terms of use, but is not the Version of Record and does not reflect post-acceptance improvements, or any corrections. The Version of Record is available online at: http://dx.doi.org/10.1007/s10617-011-9072-8

(Article begins on next page)

06 September 2026

Noname manuscript No.
(will be inserted by the editor)

Creating Portable and Efficient Packet Processing Applications

Olivier Morandi · Fulvio Risso · Pierluigi Rolando · Silvio Valenti · Paolo Veglia

Received: date / Accepted: date

Abstract Network processors are special-purpose programmable units deployed in many modern high-speed network devices, which combine flexibility and high performance. However, software development for these platforms is traditionally cumbersome due both to the lack of adequate programming abstractions and to the impossibility of reusing the same software on different hardware platforms.

In this context, the Network Virtual Machine (*NetVM*) aims at defining an abstraction layer for the development of portable and efficient data-plane packet processing applications. Portability and efficiency are achieved altogether by virtualizing the hardware and by capturing in the programming model the peculiar characteristics of the application domain.

This paper validates the NetVM model, demonstrating that the proposed abstraction coupled with a proper implementation of the NetVM Framework is able to provide *generality* (i.e., capability to support a wide range of applications), software *portability* across heterogeneous network processor architectures, and *efficiency* of the generated code, often exceeding the one obtained using state-of-the-art compilers.

Keywords Network Virtual Machine · Network processors · High-speed packet processing · Network code portability

1 Introduction

During the last decade, the increasing requirements in terms of flexibility for the design of high-speed networking devices have pushed the Industry towards the development of network processors. These programmable devices usually provide several concurrent execution units with instruction set architectures specifically targeted to packet processing, and integrate special-purpose hardware coprocessors for offloading computationaly intensive functionalities (e.g. hashing). Even though such devices may not be able to achieve the same

O. Morandi, F. Risso, P. Rolando
Department of Computer and Control Engineering, Politecnico di Torino - ITALY
Tel.: +39-348-723-06-76
E-mail: {olivier.morandi, fulvio.risso, pierluigi.rolando}@polito.it

S. Valenti, P. Veglia
TELECOM ParisTech, Paris - FRANCE
E-mail: {silvio.valenti, paolo.veglia}@enst.fr

level of performance of custom-designed ASICs, they guarantee more flexibility thanks to their programmability.

However, network processors have traditionally their Achilles' heel in the lack of a proper programming infrastructure. In order to achieve maximum performance, programmers have to deal with low-level hardware details, e.g., by explicitly accessing special-purpose coprocessors, or by manually partitioning program sub-tasks across the many available execution units. In fact, although vendors provide Software Development Kits (SDKs), these either require the use of an assembly language for some specific functions, or, if some C language flavor is available, hardware units must still be explicitly accessed through ad-hoc primitives and functions, denoting an obvious lack of abstraction. Due to the extreme variety of architectures (which span from symmetric multi-processing platforms like the Intel IXA family [16] and the more recent Cavium Octeon [28] network processors, to systolic array dataflow processors like the Xelerated X11 [33] and the Bay Microsystems Chesapeake [21]), the reuse of the same software on different hardware platforms (often even on other processors of the same family) becomes almost impossible. Applications developed and optimized for a specific Network Processing Unit (NPU) must be redesigned from scratch upon being ported to a different processor, undergoing once more the entire development cycle.

Given such high heterogeneity, the problem of defining a common programming model able to provide *generality* (i.e., capability to support a wide range of applications), *portability* (across a wide range of network processor architectures) and *efficiency* is particularly difficult. Current solutions generally focus on one or two of these aspects, but nothing exists that looks at the problem in a comprehensive manner. Particularly, generality and portability are usually not taken into account, since proposed solutions are mainly targeted to specific hardware architectures or application classes.

In such a scenario, the Network Virtual Machine (*NetVM*) [3][2] aims at applying the "write once, run everywhere" paradigm proposed by the Sun Java Virtual Machine [19] and the Microsoft Common Language Runtime [11] to the field of network processing, where performance is a key factor. The NetVM defines a virtual computing platform, based on a data-driven programming model, in which hardware is virtualized, thus hiding the quirks of the target architecture from the programmer.

One of the main objections to this approach is that the adoption of a hardware abstraction layer, while enabling portability, would result in a substantial overhead, wasting the benefits of using special-purpose and optimized hardware architectures. In this paper we demonstrate that this claim is not necessarily true in the case of a virtual machine specifically designed for packet processing applications, such as the NetVM. In particular, the NetVM model exposes a set of key features that, besides making it a good target for different high-level languages, enables both portability and efficient feature mapping, at least on four target platforms tested, namely the Intel x86 and x64 general purpose architectures, the Cavium Octeon [28] multi-core processor and the systolic-based Xelerated X11 [33] network processor.

In order to support our vision, we designed and implemented the NetVM framework, whose main component is a multi-target optimizing compiler implementing the NetVM model. Optimizations, crucial for performance, operate on two different levels: first an architecture-independent module removes redundancies and useless computations, then a set of target-specific backends performs the actual mapping between the NetVM model and the target machine, possibly exploiting the special hardware features available on the selected NPU. Experimental results will show that NetVM applications can be efficiently executed,

without any change, on the three platforms of choice, with performance often better than state-of-the-art compilers and manually optimized code.

This paper is structured as it follows. Sec. 2 summarizes the related works, Sec. 3 presents the NetVM model and Sec. 4 outlines the implementation of the framework and the general optimizer module. Even if part of the NetVM framework, the backends represent a key component for achieving the wanted objectives, and hence are presented in Sec. 5. Experimental results are reported in Sec. 6 and conclusions are drawn in Sec. 7.

2 Related Works

Recent years have seen the problem of creating a suitable framework for programming network processors being widely investigated.

Click [25] is a framework for implementing a modular router by interconnecting different packet processing modules (under the form of C++ classes) that implement specific functions (e.g., packet classification, queuing, scheduling). The interconnections between modules create a directed graph that represents the flow of packets inside the router. NP-Click [29] extends the Click programming model and maps it on Intel IXP network processors, showing that the level of abstraction introduced makes the application development easier and enables an efficient mapping on the Intel IXP1200 special-purpose architecture. Memik et al. [20] demonstrate the advantages of a modular structure in network processing applications and describe a system, called NEPAL, which is able to extract the constituting modules of a sequential network-processing program for mapping them on parallel execution units. PPL (Packet Processing Language) [27], defined by IP Fabrics Inc., is a declarative language for programming network processors of the Intel IXA family. A virtual machine executes PPL programs on the target platform and maps high-level constructs onto the available hardware features, enabling the transparent exploitation of parallel processing engines. Wagner et al. [31] propose a C compiler for an industrial NPU, showing that exposing low-level details in the language through compiler-known functions allows an efficient exploitation of the available hardware features without relying on assembly language. PacLang, by Ennals et al. [10], is a framework that provides application designers with a simple high-level language able to automatically partition packet processing programs on parallel execution units and that works on Intel IXP NPUs. Shangri-la [8] follows a more general approach and consists in a domain-specific programming language named Baker and a profile-guided compiler infrastructure, which is able to optimize and map an application onto Intel IXP NPUs. The Network Runtime Environment (NRTE) [32] is a runtime system implemented as a C library, aiming at providing a network programming environment for multicore NPUs. The programmer is in charge of defining a set of thread-safe packet processing functions that the runtime environment will map onto the underlying platform, possibly replicating some of them across the available hardware threads.

These approaches generally fail to provide a comprehensive framework for achieving both efficiency and portability across heterogeneous architectures. In particular, solutions targeted to a specific platform focus on performance and therefore tend to expose a set of primitives tied to the characteristics of the hardware in high-level programming languages, resulting in a lack of abstraction. For instance, programming models targeted to multi-core network processors often include explicit primitives for task/thread synchronization. On the other hand, others may provide means for directly accessing hardware coprocessors, for example through library functions and APIs, or language intrinsics (e.g., compiler-known functions). As such, programmers are forced to structure their software according to the

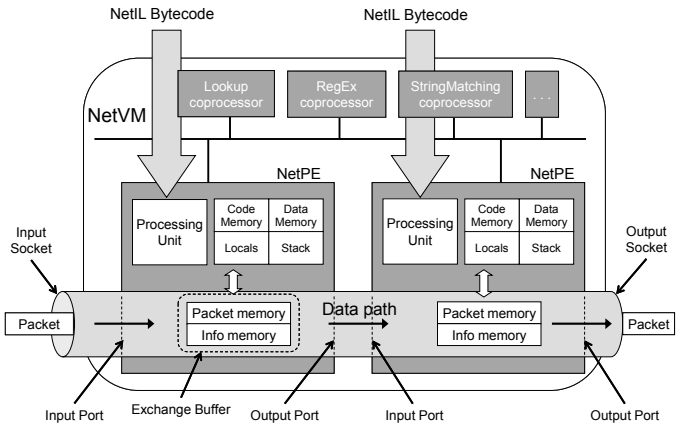


Fig. 1 NetVM Architecture

execution model supported by the hardware, e.g., by defining the appropriate task/thread partition and dealing with synchronization issues explicitly. In other words, these models are too tightly tied to a target architecture and their porting to another platform may be too costly if not impossible (e.g., thread synchronization cannot be mapped on a systolic array processor). Vice versa, approaches that are more application-oriented usually tend to completely hide the details of the underlying hardware in order to enable software portability. However, these proposals lack in flexibility, being targeted towards a specific class of applications only.

In conclusion, to the authors' knowledge, there exists no approach to packet processing software development which is able to simultaneously achieve *generality*, *portability* and *efficiency*.

3 The Network Virtual Machine

The NetVM can be considered as a system composed of the combination of a programmer-visible model and set of APIs and its implementation as a portable framework. Both portions are critical to our design goals: the definition of a powerful yet easy-to-use programming model makes the NetVM flexible and generic enough to support multiple kinds of packet processing applications without requiring exceptional programmer efforts, while an optimized implementation is required to ensure portability of NetVM applications across a wide range of platforms and provide high performance.

The issues of designing a good programming model and providing a suitable implementation are intertwined. If the model is too rich or offers complex features then the underlying implementation will have to be complex as well in order to provide all the required functionality; besides, simple hardware platforms might prove unable to provide support for every feature without incurring in excessive emulation overhead. On the other hand, a model too basic presenting only low-level abstractions would make programming harder and also increase the difficulty required for implementing an adequate compiler as less semantical information is conveyed. As it is often the case in software engineering, a delicate balance must then be achieved between the previous two conflicting aspects; the rest of this sec-

tion presents the NetVM programming model and the interactions of its programmer-visible components and describe how its design goals are achieved.

3.1 Programming model

The foremost design requirement for the NetVM programming model [3,2] is to provide the programmer with suitable abstractions for developing packet processing applications. In order to introduce this model we adopt a bottom-up approach, presenting the basic processing components first and then moving up in order to explain how we can create a rich application out of them.

3.1.1 Network Processing Element

The basic building block of any NetVM application is the Network Processing Element (*NetPE*). NetPEs are packet-oriented processors and their work units reflect this concept by containing a single network packet and associated ancillary information in the form of an *Exchange buffer* (detailed in Sec. 3.1.2).

NetPEs are programmed in a mid-level 32-bit stack-based language called Network Intermediate Language (NetIL); as an example, Fig. 2 presents an example of NetIL code and its x86 assembly counterpart referred to a simple filter that checks if the *ethertype* field of an Ethernet frame is equal to `0x0800` (i.e. if the Ethernet frame contains an IP packet). The adoption of a mid-level assembly language helps making the NetVM general enough to be independent from any specific high-level language. In fact, NetIL can be an excellent target for several high-level languages, ranging from declarative (e.g., rule based such as NetPFL [24]) to imperative ones (e.g., C).

NetIL is targeted towards data-plane network applications and hence it includes several network-oriented primitives while leaving other functionalities (e.g., floating point) out of the instruction set. Some examples can be found in bit manipulation instructions such as Cyclic Redundant Code (CRC) and hashing computations. Furthermore, some high-level operations widely used in data-plane applications are captured by the NetIL assembly in order to facilitate their effective mapping on the actual hardware architecture. The most notable example is represented by the multi-way branch (modeled after the *switch-case* construct of the C language that uses the value of a variable to select the target jump among several possible destinations), which is captured by a set of specific NetIL instructions. Finally, some even higher-level constructs are possible through *coprocessors*, which will be presented in Sec. 3.1.3.

In contrast to traditional register-based languages, NetIL is stack-based, following the design of most contemporary virtual machines (e.g., the Java Virtual Machine and Microsoft Common Language Runtime). Although stack- and register-based languages have equivalent expressiveness and although programs for the latter are likely to be more readable, the former have some practical advantages [30]. Among the others, stack-based languages can be considerably simpler to verify, e.g., checking programs for correct variable usage can be performed by statically emulating stack operations instead of tracing variable uses and definitions across multiple, potentially complex code paths. Furthermore, the compactness of the binary representation is improved because the implicit presence of an operand stack avoids specifying locations for source and destination operands.

```
; Code Segment
segment .push
.maxstacksize 10    ; define the maximum stack depth
  pop               ; discard the "calling" port
  push 12           ; push the ethertype offset on the stack
  upload.16         ; load 2 bytes at previous offset from packet memory
  push 0x800        ; push 0x800 (i.e., ip) on the stack
  jcmp.neq discard  ; if not equals jump to discard, otherwise...
  ret 1             ; return 1
discard:
  ret 0             ; return 0
ends
```

(a) NetIL code for filter "ethernet.ethertype == 0x0800"

```
; Packet buffer base in ecx
001 cmp word ptr [exc+12], 0x8  ; load packet_buffer[12:2]
002 jne 005                     ; if not equals jump to return 0
003 mov eax, 1                  ; move return code in EAX
004 ret                         ; return 1
005 mov eax, 0                  ; move return code in EAX
006 ret                         ; return 0
```

(b) Corresponding x86 code for filter "ethernet.ethertype == 0x0800"

Fig. 2 Comparing NetIL and x86 code

3.1.2 Memories

NetPEs carry a set of different, architecturally-visible memories which is structured to reflect the need of packet processing programs for accessing the current network packet, passing data to downstream modules (i.e., other NetPEs), and storing temporary and persistent information separately.

The first two memory types are *Packet memory*, which stores incoming packets, and *Info memory* that stores the execution context associated to a packet and therefore flows along NetPEs with the packet itself. Programmers can use this separate area to store information that should be consumed by later processing elements, e.g., a module can compute the starting offset of the TCP payload, while a following module can look for specific patterns starting at that location. *Packet memory* and *Info memory* taken together are called *Exchange buffer* and (as the name suggests) they represent the main mean of internal communication between different NetPEs running within the same virtual machine instance.

The lifespan of an exchange buffer is directly tied to the packet it carries: after a packet is either forwarded outside the NetVM or dropped its execution context is destroyed as well. Whenever persistent storage is required (e.g., to keep track of statistics, store tables, etc.) it is possible to use *Data memory*, a storage area private to each NetPE that is retained as long as the NetVM instance is running. On the other hand, scratchpad areas for information required only during a specific call to a NetPE handler are provided by both the execution stack and a set of directly-addressable local variables that are erased whenever a new execution context enters the processing element.

An explicit choice in designing the NetVM memory subsystem was not to provide any user-visible memory allocation and deallocation primitives: all memory segments are statically allocated, either during a dedicated NetVM instance initialization phase, or at the creation of an exchange buffer; in all cases area sizes are decided by the programmer and must be strictly enforced by NetVM implementations. This choice is mainly dictated by performance constraints, as run-time memory management can be costly on some architectures, and contributes also to reduce a relevant source of complexity and errors; in most real-world packet processing applications persistent or complex data structures (e.g., a forwarding table) are usually created by the control plane (e.g., through a routing protocol process or manual configuration) and consumed in a read-only fashion by the data-plane program, thus requiring no complex primitives for their manipulation.

The rationale behind the rich NetVM memory model is to satisfy the need perceived by the programmer to have memory areas available with just the "right" semantics, depending on the specific requirements. A single, large, shared memory area (as it is natively available on most general-purpose platforms) would make the task of the programmer harder because it might require to reimplement multiple times commonly-used operations (such as those required to handle the flow of exchange buffers), and would also impede the work of the compiler. In fact, explicitly partitioning memory accesses into different categories allows ad-hoc compilation strategies to be used, e.g., to decide data placement where multiple memory areas with different properties are available on a hardware platform, without requiring extensive code analysis. The NetVM memory model is also tailored to the network-of-NetPE execution model and it contains provisions to allow and simplify parallel implementations: as an example, no data can be (nor needs to) be shared among multiple NetPEs, as data memory is private and info memory belongs to a single NetPE at any given time, thus simplifying the automatic introduction of synchronization strategies by the compiler[1].

3.1.3 Coprocessors

A quick study of packet processing applications and their typical hardware platforms shows that it is easy to identify a common set of high-level functionalities that are commonly required and that are often implemented directly in hardware on many network processor architectures. Some examples are table lookups, either exact or following a longest prefix matching algorithm, string matching, regular expression matching, etc., all of which are often accelerated using appropriate coprocessors such as Content Addressable Memories (CAMs), Ternary CAMs (TCAMs) or other appropriate hardware.

While it is certainly possible to delegate the implementation of these functionalities to the programmer, it is more productive to encapsulate them behind a clearly-defined interface and offer them as ready-to-use components. For this purpose the NetVM exposes the concept of *Virtual Coprocessors*, abstract functional modules that interact with NetPEs as black boxes hidden behind a well-defined, platform-independent interface. Instances of virtual coprocessors can be attached to NetPEs and accessed through dedicated NetIL instructions; each coprocessor exposes sets of input and output registers where parameters can be stored and results read back, and offers a set of operations acting on interface registers and its internal state representing the desired computation. Besides simplifying application development, the availability of coprocessors also improves NetVM performance as wherever possible direct mapping to hardware primitives is employed to implement coprocessor

[1] It is worth noting that synchronization primitives are not needed when programming at the NetVM level (as detailed in Sec. 3.1.5), but may be needed when mapping the NetIL code on the the underlying hardware.

instructions; even on platforms with no suitable support it is possible to provide a single optimized implementation that can be readily reused with little effort.

In addition to the aforementioned built-in coprocessors, new ones can be defined by NetVM users and implemented with custom code; apart from their interface specification, new coprocessors do not have to abide by any rule of the NetVM model, nor require new dedicated NetIL instructions. For this reason they can be used to extend NetVM capabilities or to bypass any constraint imposed by the NetVM, as an example to provide resizable memory areas, storage that is persistent across NetVM instance restarts and more.

3.1.4 Building NetVM applications

In data-plane network applications is often possible to isolate specific operations or self-contained functional modules that perform a well-defined set of operations on a packet, then send it away, along with their results, for further processing. As an example, most packet processing applications start with simple filtering modules that discriminate between packets that can be processed (as they are destined to the specific program or machine) and those that can be immediately discarded (being malformed or not required by next modules); routing, switching or classification modules can follow and other more complex functions, such as regular expression-based payload inspection, are also often present. In order to support this scenario, it is necessary for the NetVM to provide an easy way to define and interconnect these modules, and standard interfaces for their interaction.

Being NetPEs the basic building blocks of any NetVM application, a complex NetVM application can be composed of a set of independent NetPEs, each one in charge of a specific task, connected together in a logical network; Fig. 1 shows an example of a NetVM application composed of two NetPEs interconnected in a pipeline. Each NetPE can be seen as a functional module, acting as a black box with its own private execution context, its persistent data in the data memory and user-specified programming, working sequentially on a single packet at a time. The impossibility to share persistent data across NetPEs does not represent a limitation when writing applications, since the programmer can simply place all the code that requires the same persistent data into the same NetPE. For instance, keeping persistent memory private to a single NetPE also matches the intuition that data required by a specific functional module (e.g., pattern matching) does not have to be shared outside that component in an uncontrolled fashion.

The NetVM model allows the programmer to define arbitrary topologies of NetPEs as long as they constitute a direct acyclic graph, a requirement which derives both from modeling choices and implementation considerations. With regard to the former, it makes sense to disallow cyclic networks because packet processing applications usually perform their work by following the natural header sequence of network packets, going from low layers (e.g., Ethernet headers for the MAC layer) to high layers (e.g., HTTP payloads). Once a lower-layer header is processed, all the relevant information required for later computation is usually saved in memory to be reused by latter modules that take care of other packet portions; only in specific cases (such as tunneling) multiple runs of the same algorithms might be needed. In any case, since the NetVM does allow loops to be executed within NetPE, the inability of having loops in NetPE interconnections does not restrict the formal expressiveness of the model. On the other hand, acyclic interconnections simplify the implementation of the NetVM by providing a strict guarantee that no matter what run-time path is chosen, each NetPE will be traversed at most once. In turn this assumption proves valuable for implementing NetPE interconnections (intuitively, multi-NetPE programs can always be inlined) and for supporting implementations where NetPEs are able to process multiple work units

concurrently or in an optimized fashion, as there is a guarantee that after one is completed it will never reenter the same processing element to further affect its internal status.

Each NetPE defines a set of incoming and outgoing *Ports* that are used to receive or send work units; a port can lead to another NetPE or to/from external units that are used to exchange data to/from the outside world. Those ports, global to the whole NetVM, are called *Sources* and *Sinks* and allow various external entities to be connected to the NetVM, such as physical network interfaces or *application interfaces*, enabling the virtual machine to interact also with user-defined control-plane modules. The NetVM reacts to the arrival of a packet by wrapping it with the appropriate data structures (the *Exchange buffer*), then sending it along the NetPE network.

3.1.5 The NetVM processing model: sequentiality, modularity, and data-driven execution

The NetVM processing model is build upon three pillars: sequentiality, modularity (through NetPEs), and data-driven execution.

The NetVM model is strictly sequentially in order to ensure in-order processing and delivery. In fact, it considers a single packet being processed at any given time; this design choice also simplifies programming because, being sequential, requires no specific precaution (e.g., synchronization) from the programmer. Although this completely sequential execution appears to be an unacceptable limitation, especially when considering that modern processors (either network-oriented or general purpose) are usually composed of multiple cores, we have to distinguish between the *model* (that is mainly oriented to programmability, to guarantees formal properties of applications, etc.), and its *implementation*[2]. Indeed, a sequential model does not necessarily imply a sequential implementation: on the contrary, the model expresses a set of criteria any compliant NetVM implementation must adhere to, but it does not dictate their exact behavior and deviations are allowed as long as the final result (in terms of internal state, packet content and ordering, etc.) is the same that would be computed by a completely sequential machine.

The second pillar is modularity, which is achieved by proposing a programming model that invites the programmer to split its application in multiple NetPEs, which act as elementary "processing elements". Since NetPEs limit the scope of persistent data (which cannot be seen from another NetPE[3]), they implicitly represent critical regions for parallel NetVM implementations. This is the reason why "fat" NetPEs are discouraged: the compiler could find an hard task when trying to provide an efficient parallelization of the code when instantiated on real hardware.

The third pillar is data-driven processing. Instead of a fully imperative programming model where communication between different modules is based on function calls, NetPEs are event driven and their handlers are activated upon the arrival of an exchange buffer on one of the input ports, in a way that resembles a dataflow architecture. In a similar fashion, after a NetPE has finished processing its current work unit it will either drop it (thereby causing another packet to enter the NetVM instance) or forward it on one of its outgoing interfaces which, if connected to another NetPE, will trigger its handler in cascade. Due to their flexible interconnection topology, it is possible for a single NetPE to be connected to multiple subsequent processing elements: in this case an exchange buffer will be sent only

[2] Parallel implementation of the NetVM will be presented in Sec. 5.4.

[3] Persistent data can be found also in coprocessors; however also in this case the access is strictly sequential and the data is not visible outside the coprocessor, hence having the same properties described for the data inside NetPEs.

to a single destination. In this case, exchange buffer duplication is required; furthermore the execution for the original packet is frozen until the duplicate flows out of the virtual machine, hence preserving the invariant of having at most an active execution context at a time in every NetVM instance. While simplistic, the duplication semantic currently supported is adequate in most practical cases, e.g. to support multicast or broadcast transmissions. It should also be noted that the reverse (e.g. multiple packets contending for the same NetPE) can never happen no matter what the toplogy is, as the invariant of having a single packet in transit is strictly enforced.

Finally, in addition to its processing handler, each NetPE can also define an initialization handler which is called once upon startup to initialize its persistent state, if required.

The processing model is a mixture of an imperative execution model (inside each NetPE), which is compatible with traditional, single-threaded programming paradigms, and a data-driven model (when connecting NetPEs together); together, they are designed to simplify the parallelization of NetVM applications on real hardware by appropriately restricting data sharing and control flow evolution. Having work units flow in a way that is similar to what is specified by the *dataflow* model is not a novelty in networking applications [25, 20, 8, 32], as these applications can be effectively described as a collection of relatively independent packet-processing tasks. In our case, the data-driven processing model allows a seamless migration of jobs between one processing element to the following, while sequentiality avoiding the usage of synchronization primitives in the code. Furthermore, data isolation (achieved by the NetPE-scoped persistent data memory and the exchange buffers) favors a parallel implementation.

3.2 Support for control-plane operations

The core of a packet processing application is often the portion which operates on the fast path of network devices, that is, where performance is an essential requirement.

Most applications, however, also include another part, often executed asynchronously, that takes care of tasks such as handling configuration or exceptional conditions: this portion is the control plane. As an example, in a layer-2 forwarding application (e.g., an Ethernet bridge) the data plane receives network packets and forwards them to the appropriate interface (while updating the forwarding table); the control plane is responsible for enabling or disabling interfaces to prevent loops in the global network topology, and for periodically purging stale entries (i.e., those beyond a certain age) from the forwarding table.

Requirements for an application control plane are rather different from those of its data plane: while the latter needs high processing performance, this aspect is less sensitive in the former as its routines are rarely called on critical processing paths. For this reason control-planes are most often physically implemented with separated, general purpose processors that are able to instruct and control the faster network processor (or processors) constituting the data-plane. Since general purpose processors are easily programmed in a portable fashion using vanilla languages such as *C* and since most control-planes do not perform packet processing activities, the NetVM does not provide support for control-plane operations.

3.3 Safety of NetVM applications

An important aspect of modern virtual machines is the level of isolation they offer, separating applications from the rest of the hardware and software execution environment. Such

separation is possible because of the NetVM position in the application stack: acting as a filter, it can have full control over what instructions are actually executed and how other software components (such as libraries or the underlying operating system) are accessed. The isolation provided by virtual machines is especially important in the case of packet processing applications that usually run on embedded hardware where little or no protection is offered by an operating system while, at the same time, applications are subject to a continuous stream of untrusted and potentially malicious input.

Complementing its model, the NetVM has been designed to offer a set of run-time safety provisions that prevent applications from adversely affecting their execution environment by exhausting memory, corrupting the state of other processes or misusing computational resources by getting stuck into infinite loops.

NetVM safety features are implemented through multiple means. In order to keep run-time overhead low, whenever possible safety is enforced through design provisions. As an example, acyclic NetPE interconnections have the side effect of making packet processing times finite if each NetPE can be shown to terminate; fixed-size memory areas prevent memory exhaustion problems from surfacing and allow NetVM users to calculate how much memory will be required by each application. Perhaps more importantly, NetPEs do not expose code memory to the programmer: this choice prevents self-modifiable code issues and, together with memory protection, makes VM-level programming errors such as buffer overflows unable to inadvertently overwrite NetPE instructions.

A second layer of protection is introduced by static checks that analyze NetIL programs before they are executed. Upon startup the NetVM verifies that its programs are syntactically correct (e.g., each opcode is valid and well-formed, programs terminates with instructions to either drop or forward exchange buffers, all jumps point to valid locations and so on) and also runs a large set of semantical checks designed to catch erroneous stack usage, accesses to uninitialized variables, etc.; there is also a module dedicated to catching and handling out-of-bounds memory accesses, when this is possible at such an early stage (i.e. offsets and the related memory area size are both known).

Finally, in order to handle the exceptional situations that are impossible or very hard to catch at compile-time, the NetVM is capable of automatically inserting run-time safety checks designed to handle memory bounds checking and termination enforcement. In the first case it is ensured that instructions accessing a specific memory area are always executed with a valid offset so that out of bounds accesses are prevented both beyond the NetVM instance memory space and, additionally, within the same instance different memory areas are kept isolated from one another; no hardware or operating system support is required. As for termination, it is often important that packet processing applications finish within a predefined time frame in order to prevent programming errors leading to infinite loops or abnormal consumption of CPU resources due to untested or excessively slow code paths. For this reason it is possible to enable a software watchdog mechanism that keeps track of a pre-allocated instruction budget, triggering a fatal exception when it is exceeded. Both the checks required termination enforcement and those related to memory safety are carefully placed and optimized as to incur in a very low run-time overhead.

3.4 Achieving Generality, Portability and Efficiency

The abstraction layer introduced by the NetVM exposes a set of key features which guarantee generality of the model, portability of the applications, and finally enable an efficient mapping of applications to different hardware architectures.

Generality is achieved thanks to the mid-level instruction set provided by NetIL: instead of being directly tied to any high-level programming language, it provides functionalities commonly used in packet processing applications (e.g. field comparison, bit test-and-set, multi-way branches, etc.), leaving out unneeded features.

Efficiency and portability can be achieved at the same time because of the abstraction layer the NetVM model sits on: features specific to a given target architecture are completely hidden away for portability, while at the same time the programmer can provide the compiler with all the relevant information about application semantics, thus enabling an efficient mapping. In other words, the NetVM programming model allows the programmer to specify an accurate description of his intentions through specific constructs borrowed from the application domain; while this limits the freedom of the programmer when compared to more general languages such as C, this more detailed view enables the compiler to perform more aggressive optimizations that would otherwise be inapplicable. The aforementioned abstractions are also designed to effectively model operations performed by real-world hardware platforms thus facilitating a subsequent mapping of their high-level functionalities onto specific hardware units.

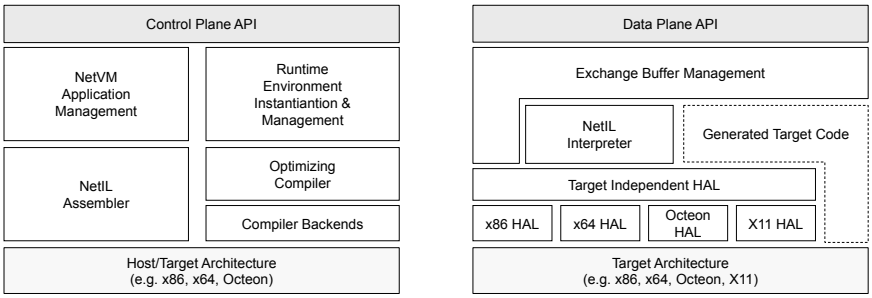
More in detail, efficiency and portability are achieved because of four main reasons. The *NetPE-based, sequential and data-driven programming model*, while avoiding the usage of synchronization primitives from the programmer, invites it to partition the application into self-contained functional units, thus explicitly describing the coarse-grained modularity and the parallelism of the application. When coming to a real parallel implementation, the modularity achievable through NetPE-based programming allows to prune away many sources of non-determinism that are intrinsic in multi-threading [18], with major advantages for both the programmer and the compiler. On the other side, the data-driven execution model is easy to understand and favors building complex applications, also thanks to the efficient inter-NetPE communication primitives based on the "moving" exchange buffer.

A *domain-specific intermediate language* presents the programmer with high-level constructs that are commonly used in packet processing applications (e.g., the multiway branch), making it possible to capture some high-level operations that can be afterwards mapped efficiently on hardware platforms with noticeable speed-ups. However, in a departure from other proposals, these details are captured by a language that is not tied to any specific hardware platform, therefore preserving portability.

A *structured memory model* closely reflects the needs of the programmer with respect to storing *(i)* temporary or *(ii)* persistent state, *(iii)* to exchange information across different application modules, and *(iv)* to access the contents of network packets. This way each memory reference acquires a semantic meaning both for the programmer and the compiler, paving the way for specific optimizations. It is worth noticing that no information is given to the programmer about which kind of hardware device should be used for mapping a NetVM memory on the target architecture. In the common case of the underlying platform offering different memory devices, the compiler that can always choose the more efficient mapping solution, therefore enabling portability while preserving efficiency.

Finally, *virtual coprocessors* enable the exploitation of advanced and common functionalities that may be present as hardware coprocessors in NPUs (e.g., lookup, string matching, etc.). These abstractions are provided to guarantee code portability through well-defined interfaces, while making it straightforward to directly use hardware devices, where available. Extensibility is also ensured, since new coprocessors can be easily added to the architecture.

Although the current implementation of the NetVM framework is still unable to offer the automatic partition of packet processing programs on multi-core architectures (in case multiple NetPEs are present, each one can be mapped on a different core, but the code of



(a) Management and Control Components (b) Data Processing Components at Run-Time

Fig. 3 Architecture of the NetVM Framework

a single NetPE cannot be partitioned automatically on multiple cores), results presented in Sec. 6 will confirm that previous speculations are correct and that the NetVM model is able to deliver high performances, while ensuring complete code portability and generality through a proper mid-level abstraction layer.

4 The NetVM Framework

The *NetVM framework* is a C library that implements the NetVM model, including a portable runtime environment, an optimizing multi-target compiler and interface functions for the creation and instantiation of application configurations. An overview of its architecture is presented in Fig. 3(a), which shows in detail the components involved in creating NetVM applications and starting their execution, and Fig. 3(b) that presents the components that perform the actual packet processing. APIs are available to control both aspects.

In Fig. 3(a), the *NetVM Application Management* module provides the functions for creating the basic entities of a NetVM application, namely NetPEs, Sockets and Connections; the *NetIL Assembler* generates bytecode for a specific NetPE from a source listing. Other functions create an application instance along with its runtime environment, while the compiler translates application code into native instructions through multiple backends.

In order to be executed on a given platform, NetVM applications require a transparently-provided runtime environment acting as an adaptation layer to hide hardware characteristics and to provide the facilities required for communicating with the external world. Some examples include I/O functions, e.g., to read packets from a network interface, augment them with the appropriate metadata (e.g., timestamps), and deliver the result to the input socket of the NetVM; coprocessor handling, e.g., the required code to exploit existing hardware modules or equivalent software implementations of unavailable components; and the capability to manage application resources (e.g., defining the appropriate memory space). These functionalities are provided by the components of the NetVM framework shown in Fig. 3(b). An ad-hoc API allows the programmer to programmatically inject packets into the application, which are converted in proper *Exchange Buffers* and sent to either the previously-compile native program or to the NetIL interpreter (that is able to directly execute NetIL code) for processing. A Hardware Abstraction Layer (*HAL*) provides the functionalities needed for correctly mapping the NetVM model onto the target architecture. On platforms with no

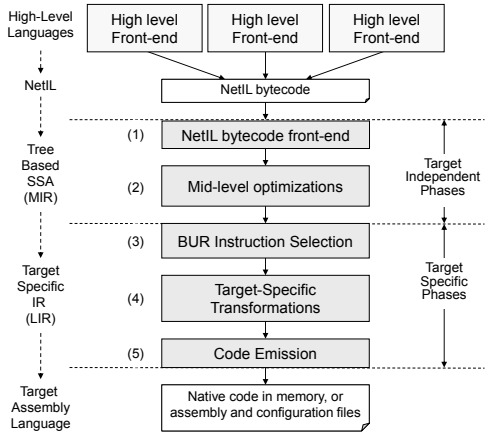


Fig. 4 Architecture of NetVM Compiler Infrastructure

hardware component suitable for a specific NetVM entity (e.g., a coprocessor), the HAL provides its software emulation.

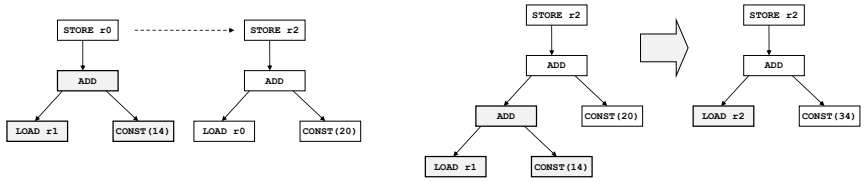
4.1 Compiler Infrastructure

The compiler follows the classical 3-stage model commonly employed in the design of multi-target optimizing compilers, as shown in Fig. 4. In order to support retargeting, most of the compilation process phases are designed to be shared to all the possible targets, while platform-specific code generation phases are isolated in different back-ends. The overall compilation process is structured as follows: (1) the compiler front-end checks the formal correctness of the source program and builds a medium-level intermediate representation (*MIR*) of the code; (2) the MIR is fed into the optimizer, that aims at removing code redundancies and improving efficiency; (3) a platform-dependent backend lowers the optimized MIR to a low level intermediate representation (*LIR*), which is very close to the assembly language of the target architecture and (4) performs additional optimizations, then, finally, (5) the resulting machine code is emitted.

A program in MIR form is a list of expression trees: their root nodes represent statements (i.e. assignment and control flow operators), while leaves contain the expression operands (e.g., constant values or registers). The LIR form, instead, represents the program as a sequence of three-address instructions, closer to the target machine language. A multi-level intermediate representation is used because of the need to delay the lowering phase, so that as much information as possible on the source program is provided to the optimizer. This enables more aggressive optimizations based on the knowledge of the semantic of the high-level constructs employed by the programmer, as will be pointed out in Sec. 5.

The whole compilation framework is designed to be modular, in order to facilitate the implementation of new back-ends. Additionally, the optimization algorithms are able to work on both on MIR and LIR, and each backend can configure the optimizer in order to apply only the transformations that are suitable for the target platform.

Depending on the selected backend, the compiler can operate either in Just in Time or in Ahead of Time mode. In the former, the target binary code is directly emitted in memory



(a) Original code (b) Optimized code

Fig. 5 Optimization of packet demultiplexing code

and executed within the same process. In the latter, the NetVM operates as a cross-compiler and the target code is emitted into a set of assembly files that can be further processed by platform-specific tools and linked with the run-time of the NetVM framework.

4.2 Mid-Level Optimizations

In order to provide a general framework for simplifying the development of dataflow analysis and optimization algorithms, the NetVM compiler translates the MIR into Static Single Assignment (SSA) form [9], where every variable is assigned exactly once. This transformation preserves program semantics while making explicit in the MIR the relationships between the definition and the uses of each variable, a fundamental requirement for many optimization algorithms.

The optimizations implemented in the NetVM framework derive from well-known techniques [26] based either on *Data-flow analysis*, which focus on the data dependencies in order to simplify or delete redundant instructions, or on *Control-flow analysis* that optimize the execution of a program by exploiting the properties of the control-flow graph. The code transformation algorithms actually implemented have been chosen after an accurate analysis of existing NetIL code, either hand-written or automatically generated through a set of high-level frontends. As an example, the code generated by the packet filter compiler described in [23], presents several redundancies and suboptimal recurrent patterns. The implemented algorithms take into account such situations and aim at removing the negative effects introduced by automatic code generation.

Among the data-flow optimization algorithms implemented in the framework, *Constant Propagation* replaces most constant-initialized registers with their respective values and often enables the application of other optimizations, such as *Constant Folding* and *Dead Code Elimination*; the former tries to simplify operations whose operands are constants by replacing them with the result computed at compile-time, while the latter removes instructions that define variables that are never used (i.e. *dead variables*). *Algebraic Simplification* has some similarities with constant folding, but, instead of computing at compile time the result of constant expressions, it exploits algebraic properties of arithmetic and logic instructions to replace whole sub-expressions that can be computed at compile time with their result: as an example it is able to replace *(a * 1)* with *(a)*. *Reassociation* joins different statement trees into deeper ones, providing larger scopes for further transformations such as Constant Folding. The role of Reassociation is evident when considering the structure of typical packet demultiplexing programs, that usually contain sequences of operations to find the offsets of protocol headers and fields in the packet buffer. Fig. 5(a) shows an example of such a

sequence of statements where a variable holding the current offset (`r0`) is incremented to point to the beginning of the TCP header: this operation is done in two steps by adding the lengths of the Ethernet and IP headers (14 and 20 bytes respectively). The reassociation algorithm joins the two statements, thus producing the tree on the left of Fig. 5(b), which allows further optimizations: constant folding can now remove the second ADD node and replace it with the corresponding result calculated at compile-time, resulting in the tree on the right.

This pattern is very frequent in our code, particularly when we generate packet filtering programs with our NetPFL compiler [23] (available as part of the NetBee library [34]), which generates the code that computes the offset of a given field as the sum of the length of all the preceding fields. Reassociation and Constant Folding, acting together, can compact most of these elementary instructions and hence are very effective in reducing the total number of instructions of the program. Furthermore, reassociation plays an important role on the X11 platform, where we can access the memory only when the packet is in some special positions in the pipeline (i.e., in the Engine Access Point blocks) and access operations can be bundled together to reduce their number. An additional side-effect of this algorithm is to reduce the number of intermediate registers needed in the program, which is particularly effective on the x86 platform where spilling is common due to the low number of architecturally-visible registers and the relatively high cost of memory accesses.

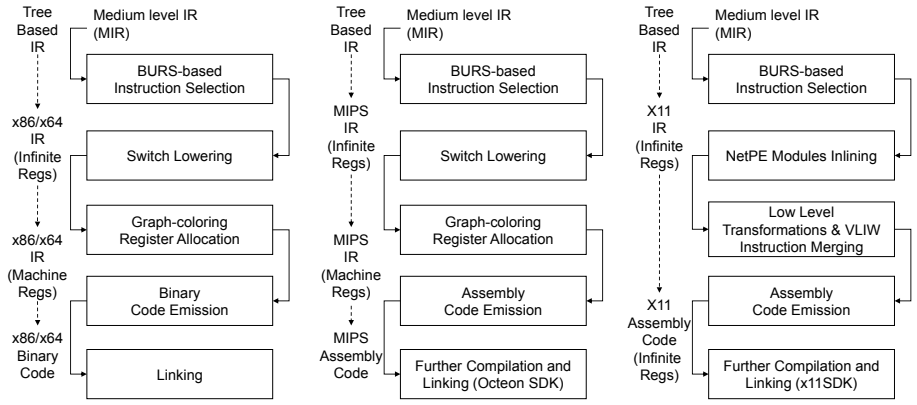
All the optimizations described above are performed on the MIR in SSA form, which is not directly executable; in order to produce actual running code, the program has to be reverted back to normal form: this step leaves the program in a state where most variables are still defined only once and a large number of copies are performed. This is suboptimal because such a great quantity of copies is cumbersome to manage and a large number of virtual registers burdens subsequent compiler modules, affecting compilation times. For these reasons we implemented a *Copy Coalescing* [6] algorithm, which scans the code for copies and tries to assign the same name to both the source and the destination variables.

Besides dataflow optimizations, the optimizer also provides algorithms that simplify the control flow structure of the program, such as *Branch Simplification*, for replacing all conditional jumps that can be evaluated at compile-time with unconditional jumps, *Jump-to-Jump Elimination* for bypassing and removing basic blocks containing only a jump instruction, and *Unreachable Code Elimination* for removing unreachable code.

Although these architecture-independent optimization algorithms are simple and widely known from classical compiler theory, they have proved to be extremely effective for two main reasons: *(i)* packet processing applications expose a very simple structure of the code, compared to general purpose ones, and *(ii)* these provide the base for further target-specific transformations that can be applied by backends, as detailed in Sec. 5. The combination of architecture-independent and target-specific optimizations results in the production of code that can be faster than the one generated by state-of-the-art C compilers, as shown in Sec. 6.

5 Compiler Backends

The NetVM framework currently implements four backends: the Intel x86 and x64 architectures, the Cavium Octeon massive multicore processor and the Xelerated X11 systolic array processor. In particular, as shown in Fig. 6, the first two share a very similar structure, while the rest, being targeted to a very special-purpose architecture, rely on a more complicated sequence of compilation phases. Every backend translates MIR statements into sequences of equivalent LIR instructions. This task is performed by a Bottom-Up Rewriting System



(a) x86 and x64 Backends (b) Octeon Backend (c) X11 Backend

Fig. 6 Compilation phases for the four compiler backends

(BURS) [13], executing a tree-matching algorithm driven by a set of architecture-specific rules that dictate how a portion of a MIR expression sub-trees can be translated into target instructions. Depending on how the source code is written, it can happen that input patterns end up split across different statements. Since the BURS operates on a single LIR expression tree at a time, pre-processing steps that rearrange subtrees into deeper structures may help creating more recognizable patterns. This is the case with the aforementioned reassociation algorithm. When a tree or subtree can be matched by multiple rules, the BURS is able to chose the combination that results in the least expensive LIR instruction sequence.

The power of BURS lies in the recognition of very specific patterns, tailoring code emission to the target platform. For instance, some portions of an algorithm can make use of native hardware components on a specific hardware platform, while traditional instructions are generated on another platform. Two examples can be found in the `LEA` opcode available in the x86 instruction set (detailed in Sec. 5.1) that replaces a bunch of NetIL instructions and the TCAM of the X11 (detailed in Sec. 5.3.2) that provides a very efficient implementation for the `switch-case` construct.

In any case, even though such techniques work well in very specific cases, their general validity still needs to be proved because they are tuned on patterns of instructions and not on algorithms: a minor reordering in the original source code might lead to different statement tree configurations that fool the system.

5.1 x86 Backend

The x86 backend follows the Just-In-Time paradigm: it generates a function in memory that includes the proper binary instructions and that receives an Exchange Buffer as an argument.

The sequence of compilation phases involved is shown in Fig. 6(a). After MIR statements are mapped onto x86 LIR instructions, a register allocation step is performed in order

to assign a machine register or a memory location to each virtual register defined in the MIR program. The register allocation algorithm implemented is based on graph coloring [14,5], using the spill heuristic proposed in [4] for minimizing spill costs and for guaranteeing an optimal utilization of machine registers.

The set of BURS rules implemented in the backend aims at addressing two problems: *(i)* the optimal exploitation of the complex instruction set of the target machine, and *(ii)* the application of packet-processing specific optimizations.

With respect to the first goal, the CISC nature of the Intel x86 instruction set enables certain complex NetIL instruction patterns to be translated into single x86 opcodes: the BURS instruction selection algorithm makes this operation straightforward. As an example, Fig. 7 presents an x86 code fragment that computes the length of the IP option fields with both its naïve and its optimized version. This value is calculated by loading the IP header field, masking it, multiplying it by four and finally subtracting 20; we can compact most of the processing into the `x86 LEA` (Load Effective Address) [4] instruction.

Non optimized	Optimized

`movzx eax, byte ptr [ebx+14]`
`and eax, 0xf`
`mov esi, 4`
`mul esi`
`mov esi, eax`
`add esi, -20`
	`movzx eax, byte ptr [ebx+14]`
	`and eax, 0xf`
	`lea ecx, dword ptr[ecx+eax*4-20]`

Fig. 7 Exploiting complex instructions in the Intel x86 instruction set

We have also implemented special rules for optimizing frequent operations of packet processing applications. For example, these often need to load a field from the packet header and compare it with a constant value: this operation is particularly expensive on the little-endian x86 processor because the standard network byte order is big-endian and a byte swap would be required. Our solution, on the contrary, uses the BURS to recognize those patterns of instructions and moves the byte swapping operation at compile time, when it can be performed by swapping the constant instead of swapping the value read from the packet buffer. A simple example is presented in Fig. 8, which shows a check to determine if an Ethernet header is followed by an IP header.

Non optimized	Optimized

`mov eax, word ptr [12]`
`shr eax, 0x10`
`bswap eax`
`cmp eax, 0x800`
	`cmp word ptr [12], 0x8`

Fig. 8 Constant byte order swapping optimization

[4] The `LEA` instruction stores in a register the effective value of a pointer that can be expressed as [base + offset * scale + displacement], where base and offset are registers, scale is an integer among 2, 4, 8, and displacement is an immediate value.

Another example is represented by the multi-way branch (similar to the `switch-case` construct of most imperative languages), which has a very sophisticate mapping on the x86 code. In fact, the back-end includes a switch lowering module that follows an approach similar to the one implemented in the LLVM compiler [17]. This technique is able to select the best mapping algorithm according to the cardinality and the density of the case set, e.g., transforming the `switch` into a set of `if-then-else` operations is the number of cases is small (e.g., ≤ 3), or using a jump table if the values on which we "switch" are almost contiguous, etc.

Finally, the x86 back-end includes a specific phase that implements an efficient linking strategy for code associated to different NetPEs: direct linking avoids returning the control to the framework when a NetPE task ends, hence reducing the overhead introduced by the runtime environment.

5.1.1 Intel x64 Backend

The compiler architecture for the Intel x64 platform (64 bits) is similar to the x86 backend and it implements all the optimizations already presented in the previous section.

The most important differences of the x64 platform consists in the extended addresses and operands (64bits registers and immediates are available), in the larger number of general purposes registers (16 against 8 available in 32 bit mode), the availability of 16 128-bit registers (formally defined for SSE instructions, the Streaming SIMD Extensions, but available to some degree also for general purpose computing) and, of course, different opcodes for the new instructions.

The x64 architecture does not provide many additional advantages compared to the x86 platform, at least for our purposes; hence the limited number of improvements compared to the previous backend. The extended range of registers makes the job of the register allocation algorithm easier, since variables have to be spilled in memory less frequently. SSE registers can also be used as additional storage but cannot be accessed directly by most traditional x86 instructions (explicit moves are required) and, under the standard platform ABI, they are also volatile across functions calls; if a system or user-provided function is used from NetPE handlers (e.g., to invoke a coprocessor that is not available in hardware or send a packet outside the virtual machine), the content of those registers may not be preserved. 64 bits operands enable to pack some 32-bit instructions in the same opcode; e.g., a check on the IP source and destination addresses (32 bits each) can be done with a single x64 instruction. However this pointed out one of the limitation of the BURS: instruction patterns are recognized only when contained in the same basic block (i.e., a straight piece of code without jumps). Being the NetVM a 32bit machine, any 64-bit compare-and-branch operation requires two basic blocks and hence this optimization has been implemented through a peephole optimization phase in the x64 compiler backend, after the Low Level Intermediate Representation (LLIR) generation done by the BURS.

Since the similarities between the x64 and x86 backends we will omit the x64 backend from the following sections; also the tests in Section Sec. 6 will only refer to the x86 platform[5].

[5] Incidentally, the performance we measured on the x64 platform were extremely dependent on the processor architecture; older CPUs usually execute x86 code faster then x64, while newer CPUs do the opposite. In any case, we never observed more than 10% improvement of x64 code compared to x86, even on our most recent machines.

5.2 Octeon Backend

Since the Octeon platform is probably less known than the x86 one, we will present a brief description of the characteristics of the processor before introducing how the NetVM model is mapped onto it.

5.2.1 The Octeon Architecture

Like most NPs, the Cavium Octeon tries to exploit the parallelism of typical packet processing applications: for this reason it features up to 16 MIPS-64 cores at frequencies up to 800 MHz. Each core has a private L1 cache, while the L2 cache and DRAM are shared. Communication primitives between cores are provided by specific hardware mechanisms; for instance, shared memories cannot be used for this tasks because a private virtual memory space is assigned to each core. The primary on-chip communication mechanism is the *work*, which is an entity created upon the arrival of a packet and queued into a specific hardware unit, the *Scheduling/Synchronization/and Order* unit (SSO). Works have many attributes that determine how the SSO dispatches them to the cores. As an example, the programmer can specify different QoS levels associated with different kinds of traffic: the unit receiving incoming packets will parse the packet header and provide a preliminary classification. The most important attribute is the *group*: in fact cores subscribe to groups and the SSO schedules works to the cores according to the subscribed groups. When a core terminates its job, it can submit the work to another group, (this, ultimately, to another core), or send the packet out to a network interface.

Besides MIPS cores, the chip also contains supporting units and coprocessors for offloading certain specific tasks. Some of these deal with the reception and the transmission of packets, some are devoted to the management of pools of memory buffers, and others implement cryptographic and string matching functionalities in hardware.

5.2.2 The Compiler Backend for the Cavium Octeon

The NetVM framework generates the code for the Octeon using the Ahead-Of-Time model and its output consists in several assembly files, C listings and configuration files that must be further processed by the Octeon SDK. The result is a native application running on the bare hardware with a minimal runtime environment. As shown in Fig. 6(b), the code generation process is not different from the x86 backend (i.e. it implements the BURS instruction selection and global register allocation), while the mapping of native hardware functionalities deserves some more discussion. More in detail, this consists in mapping the Exchange Buffer on native hardware structures and natively supporting the string matching coprocessor of the NetVM model.

With respect to the former, the Exchange Buffer can be mapped on the work structure of the SSO unit. This enables NetPEs to be distributed on different cores that communicate through the native mechanism, in a way that is completely transparent to the programmer. The general mechanism, however, is already in place and can be used in future work aiming at fully exploiting the potentialities of multi-core processing.

With respect to the second item, the NetVM model has a general string matching coprocessor that enables searching for groups of patterns in the packet payload. Patterns, initialized before the program starts, are divided into groups identified with an integer ID, so that the coprocessor can search all the patterns belonging to a group at once and return multiple matching results to the caller. While the x86 back-end provides a software implementation

based on the Aho-Corasik algorithm [1], the Octeon includes a hardware unit that is able to traverse graph-based structures representing Deterministic Finite Automata (DFA) in memory, which can be used to perform both string and regular expression matching. With respect to the Octeon processor, the DFA graph must be translated into a binary image, then loaded in a special external memory, the Low Latency Memory (LLM). During execution the cores can submit a command to the DFA engine specifying the address of the packet payload and the address of a graph in the LLM: the hardware unit automatically loads data from the packet memory and uses it to traverse the graph in the LLM to look for a match.

Finally, we note that the runtime environment for this backend is very simple and it consists of an initialization routine (automatically emitted by the compiler) to initialize processor units and instantiate the memory structure needed by the NetVM instance. The only task of the runtime environment is then to receive packets from physical interfaces and to pass them to the native functions generated by the NetVM compiler.

5.3 X11 Backend

The X11 architecture is a radical departure from general-purpose CPU designs: we will present it in a brief introduction before showing how it can be exploited to support many NetVM capabilities.

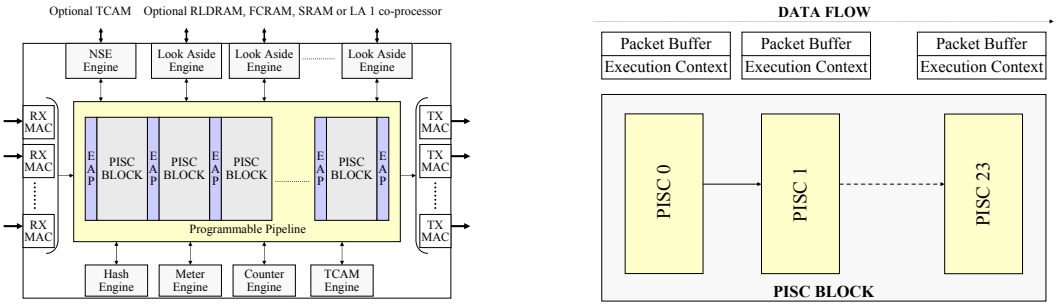
5.3.1 The X11 Architecture

The Xelerated X11 network processor is based on a systolic pipeline with a synchronous dataflow architecture, a concept shared with its predecessor X10q [7]. Figure 9 shows an overview of the X11 internal architecture. Most pipeline stages are Packet Instruction Set Computers (PISCs), simple VLIW processing units that in a single clock cycle are able to perform in parallel ALU operations, accesses to packet memory and branches. Programmers can access on-board coprocessors through multiple Engine Access Points (EAPs), specialized I/O stages located at regular intervals in the pipeline. Among the EAP-attached devices there are TCAMs, various kinds of RAMs, hashing coprocessors and more.

The whole X11 architecture is fully synchronous: packets enter the pipeline at the first stage one at a time, and each clock cycle each PISC performs a single instruction on its current data unit before forwarding it downstream and getting a new packet to operate upon from the previous pipeline stage. EAPs respect the same paradigm, accepting and completing a new operation each machine cycle. If required to absorb the latency of an external device, EAPs can be internally pipelined. As a packet traverses the pipeline, it carries an individual execution context containing packet memory, a register file, status registers, and in general all the information that constitute the complete state of a program.

Thanks to its pipelined architecture, the X11 NPU is an intrinsically parallel machine: in line of principle, at any given time a packet can be under processing in each different pipeline stage. From a programming perspective this happens automatically without explicitly parallelizing the application. Among the downsides, while the X11 architecture and its parallelism allow very fast processing rates with the hardware running at low frequency, it offers few ways to be be disabled or controlled by the programmer, and makes it hard to support applications that do not fit its design paradigm.

First, the execution contexts of different packets are totally isolated from one another; when this is undesirable, e.g., because shared tables need to be updated, the limitation can be circumvented only by using an EAP-attached device. A single device can be connected



(a) X11 Architecture (b) PISC Block Detail

Fig. 9 X11 Internal Architecture Overview

to multiple EAPs at different depths in the processor pipeline: since the processor is fully synchronous and execution cannot be stalled at any stage (in order to avoid the introduction of bubbles), concurrent accesses from multiple EAPs to shared data might be subjected to pipeline hazards such as Read-After-Write, Write-After-Write, etc. [15]. Second, the upstream-to-downstream data flow architecture makes it impossible to execute loops, and poses a hard limit on the amount of instructions that can be executed on a single packet, corresponding to the length of the pipeline[6]. As a consequence, any program containing loops must be unrolled on the X11 processor. Fortunately, many packet processing applications do not require loops (for instance, we were able to implement a complete Intrusion Detection System, presented in Sec. 6.2, without loops) and in case these are required (e.g., to handle some protocols like MPLS or IPv6 [22]) it is still possible to limit the maximum number of loop iterations in the source program to a fixed value. Therefore we expect that this issue might turn out to be not so relevant in practice at least in our target data-plane applications.

As a final, general observation, the X11 platform architecture is by design aimed at layer 2 to layer 4 processing; as such it is difficult to translate applications requiring large amounts of shared data, such as those using the NetVM data memory or stateful coprocessors. We will show how to handle simple cases (e.g., counters and read-only lookup tables) by appropriately using the external devices but others, such as dynamic TCP session tracking, are more difficult to map. On a case-to-case basis it might still possible to emulate some of these if slight modifications in application semantics are allowed: possible choices range from performing shared data updates in the control-plane context, where it can be ensured that no concurrent conflicting operations happen, to introducing more elaborate special-purpose external engines. Such solutions are beyond the scope of this paper.

5.3.2 The Compiler Backend for the X11 Architecture

The general structure of the X11 compiler backend reflects rather closely the x86 and the Octeon ones, as shown in Fig. 6(c): MIR instructions are lowered to LIR through the BURS module, then are further optimized and finally emitted ahead of time in assembly form to be processed with target-specific tools. However, given the properties of the target platform, there are some aspects and requirements that differentiate the X11 backend from the others.

[6] A packet can be made to loop transparently along the pipeline a small and predefined amount of times to emulate a larger number of stages at the expense of the throughput.

A first difference is in control flow handling. The X11 NPU lacks the primitives required to execute function calls; therefore, a NetVM application composed of multiple NetPEs must be transformed into a single compilation unit by performing an inlining step, a procedure that is always possible thanks to the acyclic NetPE interconnection graph imposed by the NetVM model. Moreover, as already noted, loops must be completely unrolled before being executed on the X11 NPU. A second critical point is the availability of different memory areas to hold the state of a NetVM application. While it is straightforward to map NetVM packet memory to the X11 packet buffer and the Info Memory and the Locals to the NPU's registers (the register file can be directly and indirectly addressed), it is much more complicated to find a persistent storage area for the NetVM data memory. The X11 NPU provides a suitable memory zone only in the external EAP-accessible RAM; however the possible kinds of operations supported are limited, mainly because the same memory area is shared across different packets that can update it concurrently, with no locking control possible. The compiler, however, having full visibility on the whole program and being in control of external unit allocation to EAPs, is able to recognize uses of the data memory that might lead to pipeline hazards and generate a warning when their absence cannot be proved. In some cases the compiler is able to avoid concurrency issues by recognizing specific access patterns and emitting atomic instructions that transparently perform complex computation directly in hardware. A case is presented later in this Section.

The X11 backend has been engineered to effectively exploit the hardware coprocessors available, both for mapping NetVM coprocessors on hardware devices, and to speed up the computation of specific NetIL instructions patterns. In the first case, NetVM coprocessor accesses are compiled into an equivalent mix of X11 instructions and EAP operations; as an example, the NetVM lookup coprocessor is efficiently mapped to the integrated TCAM module. In the second case, there might be no ad-hoc instruction to tell the compiler that a known functionality or algorithm is to be used: the BURS module is used to recognize specific patterns of instructions that can be executed with the support of specific hardware units.

We present two examples of BURS-supported pattern recognition under the form of the `switch-case` instruction and the aforementioned atomic memory accesses. With regard to the former, the NetIL `switch` opcode implements a traditional multi-way branch decision which is not directly supported by the X11 NPU. Although a naïve mapping through repeated comparisons is still possible, a large number of `case` labels would require a correspondingly large number of pipeline stages to be wasted in computing the correct destination. In order to avoid this waste of resources our backend translates a `switch` instruction into a single TCAM lookup operation: the current value (the argument of the `switch` instruction) is asked to the TCAM, which basically returns the address of the next instruction to be executed. This requires a single EAP access independently from the number of different case labels, making the switch operation extremely fast to execute and cheap in terms of pipeline resources. With respect to the second example, atomic memory accesses are the demonstration of more general capabilities. In many packet processing applications read-modify-update cycles are performed on data memory locations, e.g., to update statistic counters. NetVM does not provide a single instruction for this operation: within its sequential execution model, the programmer can simply read the value from memory, update it and write it back in discrete steps. A naïve translation of this procedure to X11 EAP memory accesses can cause the traditional concurrency faults in which some updates are lost because they are still not committed to memory when a subsequent packet causes the machine to read again the same memory. The hardware provides support for this case as it is possible to instruct an EAP to perform an atomic read-modify-update operations; the X11 backend

recognizes the relevant NetIL instruction pattern and translates it into an atomic operation, avoiding all concurrency hazards.

A further X11-specific backend module performs VLIW instruction merging. The hardware allows up to four independent operations to be executed at the same time, in order to exploit instruction-level parallelism. These can be (1) an ALU operation, (2) a move operation for copying words of up to 32 bits, (3) a load offset operation for indirectly accessing the register file or packet data, and (4) a branch instruction. Since source programs are strictly sequential, the compiler is tasked with scheduling different instructions to be executed at the same time while keeping track of data and control dependencies. Several algorithms are described in the literature for handling this task in an optimized fashion, e.g., trace scheduling [12]. The compiler currently implements a basic algorithm that works on straight-line code fragments without performing any reordering before merging. This is effective at reducing code size, even though it is a widely known result that the amount of instruction-level parallelism present in a program is greater when instruction reordering within or across basic blocks is allowed: resorting to one of the well-known, more aggressive strategies is likely to improve the emitted code quality significantly.

Finally, the NetVM model provides registers and memory locations that hold 32-bit words. This is a problem for the X11 processor that works natively on 16-bit words: 32-bit operations are possible but incur in large overheads, and sometimes it is possible to compute correct results using 8 or 16 bits only. Although this is clearly a limitation of the current NetVM model that does not explicitly support different data sizes, the X11 backend implements an heuristic algorithm that assigns the optimal, minimum size to each NetVM storage location while conservatively preserving the program semantics.

5.4 Going parallel

Although the NetVM model is intrinsically sequential, an effective implementation has to take the parallelization of the NetIL code into great consideration. Parallelization strategies may differ from one backend to another as they depend on the capabilities of the underlying hardware. For instance, the X11 processor is natively parallel and all the programs that are executed on that platform are intrinsically parallel: the compiler is aware of the hardware architecture and takes specific precautions to ensure correctly compile applications.

For the other environments, multiple parallelization strategies are possible. As a start, parallel implementations can be based either on the *pipeline* model (e.g., each NetPE is mapped on a physical core) or on the *run to completion* model, where the entire application is first inlined, then executed in multiple instances over multiple cores. Obviously, mixed modes or other more advanced strategies can be available as well.

The first strategy enables a relatively easy parallelization out of the NetVM model as long as the parallelization treats the code of each NetPE as an elementary block that must be executed on the same core. The advantage of this strategy can be found in the limited necessity of synchronization primitives due to the data-isolation properties of NetPEs and to the inter-core communication facilities provided by the exchange buffers. More care has to be used with respect to coprocessors, whose accesses have to be serialized (a simple strategy could consists in mapping those components, when not available in hardware, to a dedicated core). This model can be promising on platforms that have efficient communication primitives to send data (exchange buffers) from a core to another and to wake up the following stages in the pipeline. Those primitives are usually available in network processors (e.g., the SSO unit of the Cavium), but are currently unavailable in general-purpose processors such

as the Intel x86/x64 architecture. Although the pipeline model can be implemented also on the x86/x64 platform through proper emulation of those primitives, their lack may lead to an unacceptable overhead when executing network programs.

The second strategy requires to protect shared data from concurrent accesses. This can be easily done by the compiler thanks to the properties of the NetVM model, which mandates that all the data in shared memory is persistent and the same (potentially) applies for coprocessors data. Accesses to those resources have to be serialized through the proper synchronization methods, whose overhead may not be negligible and depend on the availability of proper primitives on the target hardware platform.

Finally, in both parallelization strategies a special care has to be taken not to violate the NetVM sequential model, since we have to guarantee that no out-of-order processing can occur (or, if this is allowed, that produces results equivalent to the ones achievable by the model). Also in this case, the potential sources of out-of-order problems are easily identifiable thanks to the properties of the NetVM model: apart from packets sent out of the NetVM, which must be delivered in the same order as they are received, we need to control the order of the accesses to shared resources (i.e., persistent data and coprocessors). In-order processing is easy to achieve on the Octeon platform, since usage of the SSO unit automatically provides this guarantee; vice versa, it must be implemented by the compiler on the x86/x64 platforms. In that case, a possible strategy consists in associating a sequence identifier to each incoming packet and by generating the proper code that enforces proper ordering when accessing shared resources and when sending packets out of the NetVM.

Parallelization capabilities in the current backends are a work-in-progress; the X11 is the most advanced due to the intrinsic capabilities of the hardware; the Cavium backend currently supports the pipeline model, while on the x86/x64 platform we implemented the run-to-completion model. In all cases the problem of dynamic core allocation and, in general, more advanced features are being considered for future developments.

6 Experimental Evaluation

The experimental evaluation of the NetVM model involves all the aforementioned major design goals (generality, portability and efficiency), which depend both on the model itself and on the quality of the implementation (i.e., the NetVM framework).

While efficiency can be evaluated objectively, it is harder to establish scientific criteria for generality and portability. This consideration withstanding, this section presents the test suite we defined to evaluate the above objectives and summarizes the obtained results.

6.1 Portability

Besides enabling efficient portability of programs that make use of the set of capabilities common across all the supported hardware platforms, a full-fledged virtual machine should also compensate for operations not provided natively. One solution to this issue consists in introducing an adequate run-time support layer, as it is done by some virtual machines such as the Java VM and Microsoft CLR.

General-purpose VMs however face an easier task compared to the NetVM because of the similarities in the architecture of different general-purpose CPUs, compared to the heterogeneity of network processors. Moreover they usually have no hard requirements concerning application performance, memory occupation or system throughput. In packet

processing, however, the ultimate requirement is getting the best possible processing speed out of the available hardware, so the advantages that derive from achieving full interoperability must be balanced against the loss of performance that this may imply. The current NetVM approach is to provide support for its model to the fullest extent the hardware allows; when this is not possible or we might suffer an extensive loss of performance, compilation is aborted. Among the possible causes of abortion there are programs requiring an amount of memory that is not available on the hardware platform, string patterns originating a DFA that exceeds the capability of the DFA coprocessor on the Octeon platform, number of the accesses to the EAPs in the X11 processor exceeding the possibilities of the pipeline, and code requiring backward jumps again on the X11 platform. Additionally, the compiler can abort in case some security checks fail, e.g., when the compiler detects some accesses to non-existing memory ranges, or the program does not terminate with instructions that either drop or forward exchange buffers, etc., as specified in Sec. 3.3.

This said, the most relevant criterion that can be used to evaluate the portability of the NetVM model consists in the number of different hardware platforms supported, currently Intel x86/x64, Cavium Octeon and Xelerated X11 processors. These platforms have been selected for being representative of different processing models and in fact they widely differ under several aspects: number and organization of processing elements, computational capabilities, availability of external coprocessors, and, at a lower level, instruction sets and memory architectures. Apart from the NetVM, there are no other development tools able to target simultaneously such a variety of platforms. Next Sections will confirm if this portability comes at the expense of other objectives such as generality of performance.

6.2 Generality

Generality can be demonstrated by showing that multiple classes of data-plane packet processing application can be effectively mapped to the NetVM architecture. We have currently implemented three major applications: a packet filter, an intrusion detection system and a layer-2 forwarder. While certainly not exhaustive, we believe that our set of test software is representative of the entire class of packet processing applications and that the capabilities and primitives exercised in our tests constitute the core of most other programs.

Packet filters are simple and well-understood; nevertheless, it is difficult to scale software-based implementations to high packet rates without giving up other features such as flexibility. Our approach is based on NetPFL compiler [23] that dynamically generates a NetVM application starting from a high-level filter statement and an external protocol database.

Intrusion detection systems (IDS) play a vital role in protecting networks from security attacks or misbehaving nodes and users. We use NetVMSnort, a tool reengineered for the NetVM from the well-known Snort IDS [22]. NetVMSnort supports many features of the original implementation and its intrinsic complexity can stress the modelling capabilities of the NetVM by employing multiple interconnected NetPEs; at the same time it also presents many implementation challenges by using advanced operations, such as string pattern matching, that benefit from proper mapping over the target hardware devices. This application works only on the Octeon and the x86/x64 backends, mostly because of memory constraints (i.e., the amount of available memory and the impossibility to access to non-aligned memory locations) of the X11 NPU. This limitation derives directly from X11 hardware design goals; in any case, it is possible to execute at least the initial classification module of NetVMSnort on the X11 NPU. This reflects a reasonable scenario where the

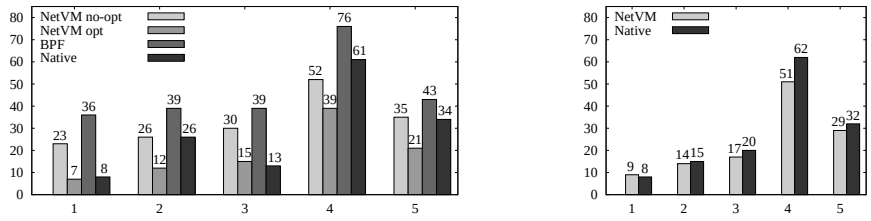
X11 performs the initial inspection at very high rates and downstream modules complete the computation.

Finally, our last test case is layer-2 forwarding, a keystone operation in packet switching networks. It can be divided into the a data-plane portion that operates on every single packet by inspecting and forwarding it, and a control-plane portion that periodically cleans the Forwarding Information Base (FIB) table up. The complexity of the data-plane portion is somewhat in between an IDS and packet filters because each packet must be examined to extract the relevant fields (as in packet filtering), then lookups are performed against persistent memory tables (the FIB) to decide the correct destination; sometimes these tables also need to be updated. As in the IDS case, this data-plane application is fully supported by both the x86/x64 and the Cavium platforms; the case for the X11 NPU deserves more attention. Particularly, we detected the (remote) possibility to generate inconsistencies when inserting a value in the X11 TCAM due to unavailability of synchronization primitives on the target platform.

In fact, the L2 forwarder application needs to insert a new MAC address in the TCAM only if the lookup for that address fails (during the bridging process), in order to avoid duplicate entries. Given this logical dependency between lookup and insertion, the X11 compiler has no chance but to implement it with 2 different EAP-operated TCAM accesses, the first one to perform the lookup and the second one to perform the insertion, if required. These 2 operations must be scheduled in 2 subsequent EAPs so there exists a small but non-negligible time interval where multiple packets with the same, yet-unknown source MAC can trigger multiple insertions, as their FIB lookup will be processed before any update can be performed. It is important to note that X11 behavior and Ethernet switching semantics ensure that the switching application behaves correctly in all cases by reverting to flooding for the small delay where no FIB entry is present even though a source MAC has been seen on the wire, then correctly sending incoming packets to the right destination. The only remaining side effect is the presence of multiple entries with the same key in the TCAM-implemented lookup table: the window of opportunity is estimated so small (3-4 packets) that in most cases no duplicates are expected at all; even when repeated insertions happen they can be pruned as appropriate by ad-hoc control-plane jobs (that are required anyway to evict expired entries).

Even though the event previously described is rare and can be solved by the control plane by cleaning up duplicate entries (Sec. 5.3.2), it is important to note these issues are detected at compile-time by the X11 backend that generates a warning message and leaves to the programmer the responsibility to evaluate their possible implications. In conclusion, it is possible in general that certain programs need specific hardware support missing from some of otherwise-supported platforms, thus preventing the compiler from translating semantically correct programs; the only thing we can do in this case is to raise a warning. These issues, however, are not due to the NetVM and cannot be avoided even when programming the target hardware platform with its native tools.

We claim that the experience resulting from our tests shows how the NetVM model can support most data-plane packet processing applications, both at the architectural level (by appropriately interconnecting NetPEs) and at the implementation level (by providing the required primitives, memory areas and control structures to write the required algorithms). Failures to do so are mostly due to unavoidable hardware limitations; in any case the compiler is able to handle them by emitting warnings or aborting the compilation in critical cases.



(a) Intel x86 (b) Cavium Octeon

Fig. 10 Filtering time for filters (ticks)

6.3 Performance

Objective run-time performance measurements can be made by running the programs on the target platforms and measuring the execution times taken to process selected inputs. While this approach is relevant for both the x86 and the Octeon NPUs, run-time tests on the X11 NPU are less significant as throughput is related to the number of instructions to be executed and not their nature, making code compactness a major factor in order to translate larger programs to fewer pipeline passes: the relevant metric for this platform is then the number of instructions emitted.

While absolute performance measurements are relevant to calculate the expected device throughput, it is also important to compare NetVM results with those from other compilation techniques. When applicable, we have written equivalent C test program compiled with Microsoft Visual Studio and GCC, for the x86 and Octeon platforms respectively; these are commercial-quality compilers that can be regarded as the state of the art. Unfortunately, no compiler exists for the X11 NPU, so in order to get a baseline for comparison we resorted to hand-optimizing assembly code following Xelerated code optimization guidelines. While obviously imperfect and error-prone, this solution nevertheless provides what we believe to be an interesting insight.

Our x86 tests were run on an 3 GHz hyper-threaded Pentium 4 processor with 2 GB of RAM; single-core experiments for the Octeon platform were conducted on a cycle-exact emulator configured to simulate a CN3080 platform, with 16 cores running at 500 MHz and 384 MB of memory. All of our test use only one core of each machine, except Sec. 6.3.5. Both platforms provide performance registers that were read using the appropriate software instructions (e.g., RDTSC for the x86 CPU); X11 data comes directly from manual assembly-level code inspection.

The rest of this Section presents an evaluation of the efficiency of NetVM-generated code, based on the performance obtained with the aforementioned packet processing applications. Some additional Sections are dedicated to some platform-specific results, which better demonstrate the flexibility and effectiveness of the NetVM compiler.

6.3.1 Packet filtering

The first test conducted consists in measuring packet filtering performance by running five packet filters[7] of different complexity. In order to get comparison baselines, the same packet

[7] Filters, according to the well-known libpcap/WinPcap syntax are `ip` (filter1), `ip src 10.1.1.1` (filter2), `ip and tcp` (filter3), `ip src 10.1.1.1 and ip dst == 10.2.2.2 and tcp src`

Table 1 Snort string matching performance on Octeon and x86

Platform	Time spent in string matching
Octeon	3.79%
x86	13.44%

filters were also created by two other generators. We chose to use the Just-in-Time version of the widely diffused BPF virtual machine; while very simple when compared to the NetVM infrastructure, it is nevertheless capable of emitting x86 machine code on the fly. Unfortunately we were unable to find an equivalent implementation on the X11 and Octeon processors, so no comparison with BPF filters is available for those platforms. A second set of test programs consists in native filters directly in the C language. Where relevant, these filters use a custom macro to speed up byte-ordering operations such as `ntoh()`, instead of relying on standard C libraries.

Results are presented in Fig. 10(a), Fig. 10(b) and Fig. 11(a). Where comparisons are available, it can be seen how the NetVM compiler is capable of generating code that is as fast as or faster than what produced by the other technologies under testing on both the x86 and the Octeon processors. As for the X11, the instruction counts are rather small, and filters fill up a reasonable portion of the pipeline, as expected.

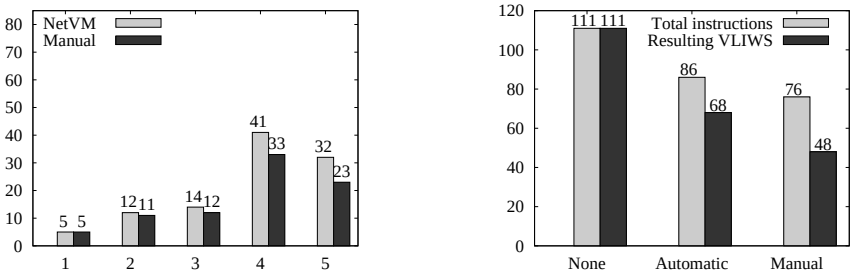
The main sources of efficiency are the intrinsic properties of the NetVM model, which exports useful information to the compiling infrastructure thus enabling very effective, albeit simple, optimizations (such as compile-time constant swapping on x86 CPUs). As it can be noted by comparing the second and third columns of Fig. 10(a), the implemented set of optimizations, although smaller than what is available in commercial compilers, is very effective at reducing execution times.

For both the Octeon and X11 NPUs, test results are good even in spite of the lack of an instruction scheduling algorithm (left to future improvements) that would prevent the processor pipeline from stalling and would improve VLIW merging, respectively.

6.3.2 NetVMSnort

We were able to successfully run the NetVMSnort application [22] on both the x86 and the Octeon platforms. Unfortunately a direct comparison with the original Snort IDS is unfeasible because our implementation uses different algorithms to process packets. Nevertheless, it is important to note that the NetVM model enables the efficient exploitation of native hardware features on platforms in which these are available. As an example, the hardware DFA unit of the Octeon NPU is used in NetVMSnort to perform string matching; as reported in the second column of Tab. 1, this greatly reduces the amount of time spent in this module when compared to platforms that must execute the whole application in software. Tests were performed on both the x86 and Octeon platforms, by configuring the NetVMSnort application with a ruleset containing 1389 rules, 1282 of which needing deep packet inspection functionalities (i.e. string and regular expression matching), and by measuring the total time needed for processing a trace containing about 10M packets captured on a real network, as well as the time spent only in the string-matching module.

`port 20 and tcp dst port 30` (filter4) and `ip src 10.4.4.4 or ip src 10.3.3.3 or ip src 10.2.2.2 or ipsrc 10.1.1.1` (filter5). The test packet was created so that filtering code was executed entirely before returning to the caller.



(a) Filters (b) IDS

Fig. 11 Instruction counts for the X11 processor

Although the X11 processor is unable to run the full application (Sec. 6.2), we extrapolated the initial packet classification module from NetVMSnort and run it on the X11 processor, simulating the deployment of the X11 as a inline traffic pre-processor. Results, reported in Fig. 11(b) are encouraging: even with a prototype compiler the performance obtained with automatic optimizations are within 20% from what can be obtained by manually optimizing the code. Moreover, even the simple VLIW merging strategy implemented proved to be rather effective in this case, as it can be seen by comparing the generated instructions with the number of resulting VLIW words; a more robust algorithm is likely to provide even better results. The differences between manual and automatic optimizations can be mainly ascribed to the simplistic VLIW merging algorithm employed and to some missed copy folding opportunities. Both these issues can be addressed with standard techniques described in literature that do not require a redesign of the compiler framework to be implemented.

6.3.3 Layer 2 forwarding

The last test conducted is a layer-2 forwarding application, which consists in a single NetPE that implements data-plane Ethernet switching in addition to the backward learning algorithm. The implementation mainly revolves around the NetVM exact lookup coprocessor, used as an associative memory to retrieve the output port set associated with packet MAC addresses. The coprocessor is used multiple times per program invocation, making its efficient implementation a must.

The application can be successfully executed on every supported platform, although with some potential issues (as previously reported) on the X11 NPU. Experimental results, reported in Fig. 12, clearly show that the code emitted by the NetVM compiler for the x86 and Octeon platforms is of comparable performance to the native implementation of the test program. Tests were conducted by processing a real-world packet capture containing about 2M packets (captured on a switch of our campus network) and measuring the time spent in executing the code of the processing element.

While on the x86 and the Octeon platforms the coprocessor implementation is entirely software, on the X11 the coprocessor implementation uses the integrated TCAM device. Further experimental results confirm that almost the 60% of time is spent performing lookup operations, that in software implementations require also the computation of an hashing function. This fact also helps explaining the very low instruction count reported in Tab. 2 for the X11 platform, where most work is offloaded to the hardware coprocessor; the resulting

Table 2 X11 instruction counts for the layer 2 forwarding application

NetVM	Manual optimizations
48	39

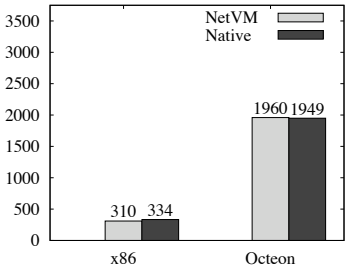


Fig. 12 L2 forwarding performance on x86 and Octeon (ticks per packet)

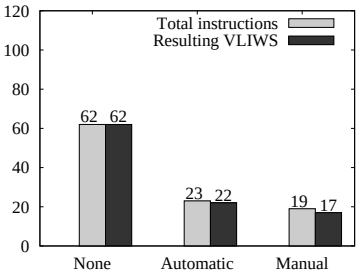


Fig. 13 X11 TCP filter (instruction count)

program is very efficient, being able to run at full wire-speed (the current generation of the X11 processor supports 4x10Gbps Ethernet links).

6.3.4 X11-specific considerations

While the previously described tests provide a good coverage of the capabilities required by the NetVM compiler, in the case of the X11 processor it is also required to test the ability of recognizing instruction patterns as described in Sec. 5. In order to do so we have modified a packet filter that recognizes TCP packets directed towards port 80 to count the number of matches, a task performed by a separated PE that is inlined at compile time.

In this example we use data memory to permanently store the total number of matches and the resulting code, if naïvely compiled, is prone to a race condition where the update from one of two subsequent matching packets is lost. The NetVM compiler is able to detect the instruction pattern related to the memory access and compile it properly into a single atomic memory operation, thus proving that the compiler infrastructure together with the combined effects of optimizations can handle at least simple cases. The test results are shown in Fig. 13.

6.3.5 Exploiting processor parallelism

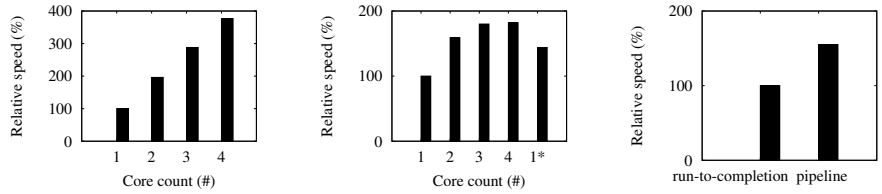
All the results presented so far were obtained using only a single core on the target platform. In a more realistic scenario multiple cores of the same processor are assigned to the NetVM to speed up execution times by processing multiple packets in parallel; performance tests have been repeated using multiple cores on both the x86 and the Octeon platforms in order to have a first insight about the feasibility and scalability of parallel NetVM implementations.

It must be noted that while the NetVM model is designed to simplify concurrent implementations, the problem of optimally allocating processing resources to different subtasks (in the NetVM case, NetPEs) is much harder to solve and is likely to be platform-dependent. At the time of this writing our NetVM implementation is limited to statically allocating or replicating NetPEs across concurrent execution units using a platform-specific strategy, as explained in section Sec. 5.4. For instance, while the X11 implementation is intrinsically parallel, the x86 version adopts the run-to-completion model and the Octeon arranges the multiple execution stages in a pipeline. The two different execution models, run-to-completion vs. pipelined, offer different performance: while the former depends on the effectiveness of its load balancer and the efficiency of the synchronization primitives required to access data shared by multiple NetVM replicas, the latter is constrained by the latency of the slowest stage.

Fig. 14 reports some preliminary results on multicore implementation of the NetVM. Experiments on the x86 platform were carried out on a 32-bit, 4 core Xeon system with 4 GB of memory when processing a 10M packets trace, while Octeon experiments were based on the same platform already used in previous tests. The relative performance shown in the graphs is the ratio between the time required to process our trace when N cores are used and the one required with a single core on the x86 platform, while it refer to the relative throughput between the pipeline and the run-to-completion models on the Octeon. This number is representative of the scalability of the approach, as we expect that in the ideal case the relative throughput is directly proportional to the number of cores allocated.

Fig. 14(a) and (b) give an insight of the run-to-completion model implemented on the x86 platform. Fig. 14(a) refers to a stateless packet filtering program and shows that, as expected, performance scale almost linearly with the number of cores dedicated to the processing. This is due to the stateless nature of packet filters that do not use any shared resource; only a small overhead is required to perform load balancing and packet dispatching. Vice versa, the Fig. 14(b) shows the results obtained when running NetVMSnort that, being stateful, requires a synchronized access to shared resources. Results demonstrate that this impairs the scalability with respect to the number of cores: four cores achieve a throughput that is only 1.8 times higher than one core and, perhaps even most important, the throughput tend to saturate even with such a small number of cores. However, we are confident that those numbers could be improved in the future when a better implementation will be available. In fact, we noticed that the synchronization primitives we use are very expensive, as shown by the the bar labeled *(1*)* in Fig. 14(b) that refers to a NetVM instance with the synchronization primitives turned off, which performs 1.43 times faster than the same implementation with synchronization enabled. We speculate that this is due to the necessity to launch concurrent NetVM instances in different processes due to some limitations in our current code, thus requiring synchronization primitives that operate at the process level that are sensibly more expensive than the ones that operate at the thread level.

Fig. 14(c) reports also the speedup of a parallel implementation on the Octeon, using the pipeline model. Results are in line with our expectations, although the advantage of using 9 cores (equal to the number of NetPEs defined in that application) seems limited as the



(a) Stateless filters on x86 (b) NetVMSnort on x86 (c) NetVMSnort on Octeon

Fig. 14 Multicore scalability

throughput increases only 1.55 times compared to the single core case. The reason is due to the limits of out current implementation of the pipeline model, whose theoretical speedup is capped the performance of the "fattest" NetPE. In our case this is represented by a module that accounts for about 60% of the total time of the application, hence representing a severe bottleneck in our application. A clever implementation of the pipeline model is left for future work.

7 Conclusions

The NetVM model has been proposed as a way to achieve generality, portability and efficiency in packet processing applications. This paper aims at validating these claims and our results demonstrate that the virtual machine paradigm is applicable also to packet processing applications without affecting their performance, and greatly improves their portability. Our compiler enables the execution of NetIL code on different architectures (four of them have been tested); it can support different classes of applications and the resulting performance may be even better than those achieved by other competing development platforms or by handwritten code.

These results can be achieved thanks to the characteristics of the NetVM model and to the quality of the NetVM framework, whose main components, an optimizing multi-target compiler and a run-time system implementing the NetVM model, allow the exploitation of the hardware features available on real network processors without affecting the portability of the generated code.

Result are encouraging, although we recognize that the implementation can still be further improved. Relevant topics include studies on possible medium-level optimizations for packet processing applications, and the possibility to fully exploit multiprocessor capabilities of NPUs with more advanced parallelization strategies.

Acknowledgements The authors wish to thank all the people who were involved in this project, particularly the many students who contributed to the development of the NetVM framework, and all the (former) colleagues who participated in the early days of this project, particularly Mario Baldi, Loris Degioanni and Gianluca Varenni who were part of the group of people who started the NetVM project back in 2002.

References

1. A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975.

2. M. Baldi and F. Risso. A framework for rapid development and portable execution of packet-handling applications. In *Proceedings of the 5th IEEE International Symposium on Signal Processing and Information Technology, 2005*, pages 233–238, December 2005.
3. M. Baldi and F. Risso. Towards effective portability of packet handling applications across heterogeneous hardware platforms. In *Proceedings of the 7th Annual International Working Conference on Active and Programmable Networks*, November 2005.
4. D. Bernstein, M. Golumbic, y. Mansour, R. Pinter, D. Goldin, H. Krawczyk, and I. Nahshon. Spill code minimization techniques for optimizing compliers. In *PLDI '89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 258–263, New York, NY, USA, 1989. ACM.
5. P. Briggs, K. D. Cooper, and L. Torczon. Improvements to graph coloring register allocation. *ACM Trans. Program. Lang. Syst.*, 16(3):428–455, 1994.
6. Z. Budimlic, K. D. Cooper, T. J. Harvey, K. Kennedy, T. S. Oberg, and S. W. Reeves. Fast copy coalescing and live-range identification. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 25–32, New York, NY, USA, 2002. ACM Press.
7. J. Carlstrom and T. Boden. Synchronous dataflow architecture for network processors. *IEEE Micro*, 24(5):10–18, 2004.
8. M. K. Chen, X. F. Li, R. Lian, J. H. Lin, L. Liu, T. Liu, , and R. Ju. Shangri-la: achieving high performance from compiled network applications while enabling ease of programming. In *In Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005.
9. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.
10. R. Ennals, R. Sharp, and A. Mycroft. Task partitioning for multi-core network processors. In *In Proceedings of the International Conference on Compiler Construction (CC) 2005*, 2005.
11. European Computer Manufacturers Association. Common Language Infrastructure (CLI) - Partitions I to VI. International standard; ECMA-335 ISO 9660: 1988 (E), ECMA International, Geneva, June 2006.
12. J. A. Fisher. *Trace scheduling: a technique for global microcode compaction*, pages 186–198. IEEE Computer Society Press, Los Alamitos, CA, USA, 1995.
13. C. W. Fraser, R. R. Henry, and T. A. Proebsting. Burg: fast optimal instruction selection and tree parsing. *SIGPLAN Not.*, 27(4):68–76, 1992.
14. L. George and A. W. Appel. Iterated register coalescing. *ACM Trans. Program. Lang. Syst.*, 18(3):300–324, 1996.
15. J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Francisco, CA, USA, 2006.
16. E. J. Johnson and A. R. Kunze. *Ixp2400-2800 Programming: The Complete Microengine Coding Guide*. Intel Press, 2003.
17. A. Korobeynikov. Improving Switch Lowering for The LLVM Compiler System. In *Proceedings of the 2007 Spring Young Researchers Colloquium on Software Engineering (SYRCoSE'2007)*, Moscow, Russia, May 2007.
18. E. A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.
19. T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
20. G. Memik and W. H. Mangione-Smith. Nepal: A framework for efficiently structuring applications for network processors. In Proc. of the Second Workshop on Network Processors (NP), Anaheim, CA, Feb. 2003.
21. B. Microsystems. Chesapeake network processor. Mar. 2007.
22. O. Morandi, G. Moscardi, and F. Risso. An intrusion detection sensor for the netvm virtual processor. In *Information Networking, 2009. ICOIN 2009. International Conference on*, pages 1–5, Jan. 2009.
23. O. Morandi, F. Risso, M. Baldi, and A. Baldini. Enabling flexible packet filtering through dynamic code generation. In *Communications, 2008. ICC '08. IEEE International Conference on*, pages 5849–5856, May 2008.
24. L. Ciminiera, M. Leogrande, J. Liu, O. Morandi, and F. Risso. A Tunnel-aware Language for Network Packet Filtering. In *IEEE Globecom 2010 - Next Generation Networking Symposium*, Miami, Flo (USA), December 2010.
25. R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek. The click modular router. *SIGOPS Oper. Syst. Rev.*, 33(5):217–231, 1999.
26. S. S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.
27. G. Myers. Overview of ip fabrics' ppl language and virtual machine, white paper.

28. C. Networks. Octeon network processors. Sep. 2004.
29. N. Shah, W. Plishker, K. Ravindran, and K. Keutzer. Np-click: A productive software development approach for network processors. *IEEE Micro*, 24(5):45–54, 2004.
30. Yunhe Shi, Kevin Casey, M. Anton Ertl, and David Gregg. Virtual machine showdown: Stack versus registers. In *ACM Trans. Archit. Code Optim.*, Volume 4, Issue 4, January 2008.
31. J. Wagner and R. Leupers. C compiler design for an industrial network processor. In *LCTES '01: Proceedings of the ACM SIGPLAN workshop on Languages, compilers and tools for embedded systems*, pages 155–164, New York, NY, USA, 2001. ACM.
32. B. Wun, P. Crowley, and A. Raghunath. Design of a scalable network programming framework. In *ANCS '08: Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, pages 10–18, New York, NY, USA, 2008. ACM.
33. Xelerated. Xelerator X11 network processor. Oct. 2003.
34. The Netgroup at Politecnico di Torino. The NetBee library. Available online at http://www.nbee.org. Aug. 2004.