POLITECNICO DI TORINO
Repository ISTITUZIONALE

Design and implementation of a framework for creating portable and efficient packet-processing applications

Original
Design and implementation of a framework for creating portable and efficient packet-processing applications / Morandi, Olivier; Risso, FULVIO GIOVANNI OTTAVIO; Valenti, S; Veglia, P.. - (2008), pp. 237-244. (Intervento presentato al convegno International Conference on Embedded Software tenutosi a Atlanta, GA, USA nel 19-24 October 2008) [10.1145/1450058.1450091].

Availability:
This version is available at: 11583/1855296 since:

Publisher:
ACM Association for Computing Machinery

Published
DOI:10.1145/1450058.1450091

Terms of use:

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

Publisher copyright

(Article begins on next page)

17 April 2024

Design and Implementation of a Framework for Creating Portable and Efficient Packet-Processing Applications

Olivier Morandi, Fulvio Risso
Dipartimento di Automatica e Informatica
Politecnico di Torino
{olivier.morandi, fulvio.risso}@polito.it

Silvio Valenti, Paolo Veglia
INFRES
TELECOM-ParisTech
{silvio.valenti,paolo.veglia}@enst.fr

ABSTRACT

It is a common belief that using a virtual machine for portable executions of data-plane packet-processing applications would introduce too many penalties in terms of performance, because of the assumed overhead caused by the presence of a hardware abstraction layer. Even if common sense proves true in the case of general purpose virtual machines, such as the JVM and the CLR, it may be wrong in case of a special-purpose network-oriented virtual machine. This paper describes the architecture of a run-time environment and a compiler infrastructure for the Network Virtual Machine (NetVM), showing that the portability of packet-processing programs can be achieved without additional penalties even over heterogeneous platforms. Our implementation supports three different target architectures: one with a general purpose processor (Intel x86), one with a multi-core network processor (Cavium Octeon) and one with a systolic-array network processor (Xelerated X11), and shows that the NetVM model (*i*) is able to abstract such heterogeneous platforms and (*ii*) enables the exploitation of hardware functionalities provided by the specific architecture; finally, it demonstrates that the performances of NetVM programs compiled into native code are comparable to those obtained using commercial general purpose compilers.

Categories and Subject Descriptors

C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems - *Real-time and embedded systems*

General Terms

Performance, Experimentation.

Keywords

High-speed packet processing, Network Virtual Machine, Network code portability, Network processors.

1. INTRODUCTION

In order to manage the ever increasing speeds of today's networks, the deployment of efficient packet-processing applications requires the use of ad-hoc hardware, such as ASICs and Network Processors (NPs). While the formers are based on extremely rigid designs, the latter have the great advantage of being software programmable. Nevertheless the lack of a common standard for both architectures and programming interfaces makes it difficult to deploy portable and efficient applications on different platforms.

With respect to the field of general purpose computers, a solution to a similar problem was the introduction of virtual machines, i.e. an abstraction layer between the user code and the hardware which enables the paradigm "Write once, run everywhere." The Network Virtual Machine (NetVM) [1][2] aims at applying such an approach in the field of network processors, where performance is a key factor. One of the main objections to this approach was that the introduction of such an additional 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, like the NetVM. In fact, NetVM programs can be executed quite efficiently, without any modifications, on three different platforms such as the Intel x86 general purpose architecture, the Cavium Octeon [3] multi-core processor and systolic-based Xelerated X11 [4] network processor.

In order to obtain good performance, we implemented a multi-target optimizing compiler infrastructure which is able to generate native or assembly code depending on the target platform. Optimizations work on two different levels: the higher level is architecture-independent and operates on the code removing redundancies and useless computations, whereas the lower is target-specific and performs the actual mapping between the NetVM model and the target machine. It is also in charge of exploiting the special hardware units available on modern NPs.

Experimental results show that the proposed approach is quite efficient: our compiler often generates code whose performance are better than the ones obtained from hand-written code and compiled with commercial general-purpose compilers.

This paper is structured as it follows. Section 2 summarizes the related work, Section 3 gives an overview of the NetVM model and Section 4 outlines its implementation. Section 5 describes the optimizer module, whilst the structure of the Intel x86 and Cavium Octeon back-ends is presented in Section 6. Experimental

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EMSOFT'08, October 19–24, 2008, Atlanta, Georgia, USA.
Copyright 2008 ACM 978-1-60558-468-3/08/10...$5.00.

results are reported in Section 7 and conclusions are drawn in Section 8.

2. RELATED WORK

The programmability of network processor architectures is a topic that has been widely discussed in studies. In [5] a C compiler for an industrial network processor was proposed showing that exposing low level details in the language through intrinsics and compiler known functions allows an efficient exploitation of the available hardware features without relying on assembly language. On the other hand [6][7][8][9] present novel domain-specific languages, programming models and compilers to address the difficult task of automatically partitioning packet-processing applications on multi-core based network processors. The proposed solutions are very target-specific because they tend to expose the features available on the target hardware to the programmer.

The solution proposed in [8] is based on a new packet-processing language and a compilation framework for optimizing programs by using profiling information gathered at runtime. The compiler also implements some optimizations that are specific to packet-processing applications, yet still aims at exploiting the characteristics of a single platform (i.e. Intel IXP 2400).

The Network Virtual Machine (NetVM) [1][2] instead aims at enabling the portability of packet-processing applications without sacrificing performance. Hence it differs from previous solutions because it provides an abstraction layer based on a dataflow programming model in which hardware is virtualized, with the result of completely hiding the target architecture from the programmer, while still allowing an efficient mapping. In addition, our model, although not directly addressing the problem of automatically partitioning programs on multi-core architectures, is designed to support a variety of heterogeneous platforms and at the same time to enable target specific low-level optimizations. Moreover our work is quite novel in taking into account the massive presence of specific hardware modules in modern NPs, allowing their exploitation but hiding their details from the programmer.

3. THE NETWORK VIRTUAL MACHINE

The Network Virtual Machine (NetVM) is an abstract processing architecture targeted at network data-plane applications. Its design aims at facilitating the programming and the deployment of different kinds of data-plane processing devices, most notably network processors, allowing a single application to run efficiently on heterogeneous hardware platforms, while providing a layer for hiding their differences.

The NetVM architecture is modular and configurable by the programmer. An application is composed of the interconnection of a set of modules, called NetPEs, which represent different functional entities that perform specific tasks on incoming packets. The execution of a NetVM application is data-driven and starts upon the arrival of a packet, which flows through subsequent NetPEs. The packet buffer is wrapped in a more complex entity called the Exchange Buffer, which also contains additional information (i.e. the *info partition*) used for transferring structured data between consecutive NetPEs. Every NetPE can be viewed as a stack-based virtual processor with a packet-oriented instruction set; it provides a set of private registers for holding temporary values and a memory for storing a persistent state that is local to the module. It also has access to the current exchange buffer, where the packet and the info partition are viewed as separate memory segments.

The NetVM model does not define any high-level programming language; instead, it defines a mid-level abstraction layer called Network Intermediate Language, NetIL, which can be employed as a target for several high-level programming languages, either declarative (e.g. rule-based packet filtering and classification languages) or imperative (e.g. C-like languages). This allows the NetVM to be general enough to support several classes of packet-processing applications (possibly written with different languages), while still enabling the generation of efficient code. Figure 1 presents an example of NetIL code and its x86 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).

NetIL code for filter "ethernet.type == 0x0800"

```
; 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 from the packet memory
  push 0x800          ; 0x800 = ip
  jcmp.neq discard    ; if not equals jump to discard, otherwise...
  ret 1               ; return 1
discard:
  ret 0               ; return 0
ends
```

X86 code for filter "ethernet.type == 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
```

Figure 1. Comparing NetIL and x86 code.

Since packet-processing applications usually rely on a set of functionalities that are often implemented directly in hardware on many network processor architectures (e.g., Content Addressable Memories for fast table lookups, hashing, string matching, etc.), the NetVM model includes the concept of virtual coprocessors, i.e. a way to make such features available to the programmer through a well-defined interface. A coprocessor is viewed by the application as a black box providing specific operations. While its coherent interface guarantees the portability of the software on different platforms, its implementation varies from platform to platform. In particular, on architectures that do not provide any hardware acceleration, coprocessors are emulated by software, while on architectures providing special purpose features, coprocessors may be mapped directly on hardware.

4. THE NETVM FRAMEWORK

The NetVM model requires a runtime environment acting as a communication layer with the external world. Its main function is to provide I/O facilities, to handle the coprocessors implementation (hardware or software) and to manage the application's resources, e.g. memory allocation. In fact a NetVM application needs to receive packets from input interfaces and to

forward them to output interfaces after the processing. Such operations are heavily dependent on the hardware characteristics. In other words, the runtime environment must implement an abstraction layer making all such details transparent to the application and to the programmer.

On the other hand, since a NetVM application relies on different elements (NetPEs, coprocessors, etc), whose configuration can be chosen by the programmer, the runtime environment has to (1) allow the programmer to create and configure each component, and (2) implement these elements on different architectures either by exploiting hardware modules or by supplying software implementation of unavailable components.

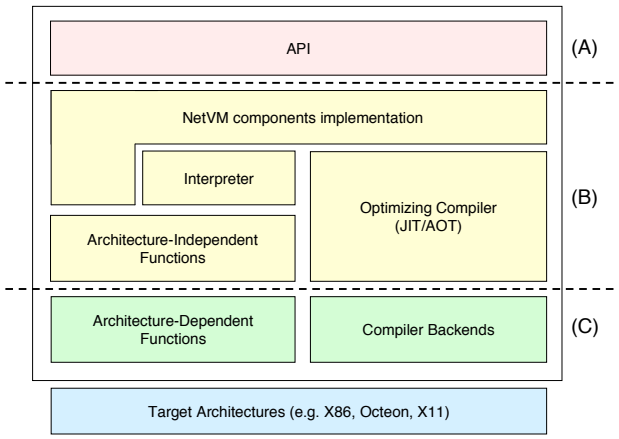

Figure 2. NetVM Framework Architecture

The NetVM model is implemented as a framework, (whose logical layout is shown in Figure 2), which comprises a portable runtime environment and a multi-target optimizing compiler. At the top of the framework (A) sits a programming interface that allows the programmer to instantiate and manage the main NetVM components in the user applications. The middle layer (B) represents the core of the framework, implementing the architecture-independent parts of the runtime environment and a NetIL interpreter, as well as the target-independent components of the compiler. Finally, at the bottom of the structure (C) we find target specific modules, i.e. the compiler back-ends and the architecture-specific parts of the runtime environment, which implement the actual mapping of the NetVM functionalities (i.e. instruction set and virtual coprocessors), on the target architecture.

4.1 Compiler Infrastructure

As Figure 3 shows, our compiler follows a classical 3-stage model. First, the compiler front-end builds a medium-level intermediate representation (*MIR*) of the source program, while checking its formal correctness; then the MIR is fed into the optimizer, whose objective is to reduce code redundancies and improve efficiency. A platform-dependent back-end lowers the optimized MIR to a low level intermediate representation (*LIR*), which is very close to the assembly language of the target architecture and performs additional optimizations. Finally, the resulting machine code is emitted.

A program represented in MIR form is described as a list of expression trees, whose root nodes represent statements (i.e. assignment and control flow operators), while leaf nodes represent the operands of an expression (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. The reason for implementing a multi-level intermediate representation is based on the need to delay the lowering phase and to provide as much information as possible on the source program to the optimizer. This makes it possible to perform more aggressive optimizations, based on the knowledge of the semantic of the constructs employed by the programmer, as will be pointed out in Section 6.

The whole compilation framework is designed in a modular fashion, in order to ease the task of adding new back-ends. In particular, the analysis and optimization algorithms are able to work on different intermediate representations, and each back-end can configure the optimizer in order to apply only the transformations that are suitable for the target platform.

The compiler can generate either machine code in memory, following the Just-In-Time paradigm, or assembly files as an Ahead-Of-Time compiler. In the latter case, the programs generated by the compiler are assembled by using third party tools (e.g. GCC or the development tools provided for the specific target platform).

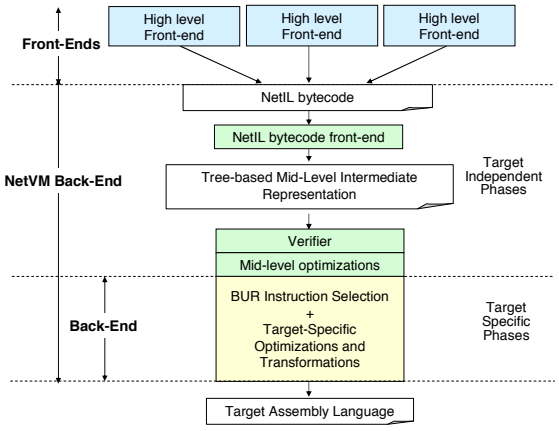

Figure 3. Compiler Architecture

5. OPTIMIZING NETVM CODE: THE FRONTEND

A reliable optimization framework is required to enable programmers to concentrate on high-level functions instead of the details of the target platform. Moreover, an efficient optimizer makes it easier to develop high-level language front-ends, which can generate redundant instructions and rely on the subsequent optimizer modules to produce efficient code.

In order to provide a general framework for simplifying the development of dataflow analysis and optimization algorithms, our compiler translates the MIR into a Static Single Assignment form (SSA) [10]. The SSA form implies that every variable is assigned exactly once, in this way the relationships between the definition and the uses of every variable are made explicit in the MIR, without altering the semantics of the program. The optimizing algorithms benefit from this form in terms of simplicity of implementation.

The optimization algorithms implemented in the NetVM framework have been selected after an accurate analysis of existing NetIL code, either hand-written, or automatically generated through a set of high-level frontends. In particular, the code generated by the packet filter compiler presented in [11], exposes several redundancies and suboptimal recurrent patterns. The implemented algorithms aim also at taking into account such situations, by removing the negative effects introduced by automatic code generation.

Among the implemented optimization algorithms, *Constant Propagation* replaces every use of constant-initialized registers with the respective values. Such optimization removes assignment instructions where a constant is copied into a register whose value is never changed and often enables the application of other optimizations, such as *Constant Folding* or *Dead Code Elimination*. The former of these tries to simplify all the operations whose operands are constant, by replacing them with the result computed at compile-time. The latter removes instructions defining variables that are no longer used later in the code (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 mathematical and logic instructions to replace sub-expressions that can be computed at compile time with their result, for example by substituting the expression (a * 1) with (a). *Reassociation* is a technique that joins different statement trees into deeper ones, enabling further transformations to be applied by other algorithms like Constant Folding [12].

The role of reassociation is evident when considering the structure of packet demultiplexing programs automatically generated by the packet filter compiler frontend. These programs usually contain sequences of operations for finding the offsets of both protocol headers and fields in the packet buffer. Figure 4A shows an example of such a sequence of statements for incrementing a variable holding the current offset (i.e. `r0`), in order to point to the beginning of the TCP header. The increment is made 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 resulting in the statement on the left of Figure 4B, allowing further optimizations. Indeed, constant folding can remove the second ADD node, resulting in the tree on the right. Since this kind of pattern is very frequent, reassociation is very effective in terms of performance gain.

All optimizations described above are performed on the IR in SSA form, but in order to produce executable code, this has to be reverted back to a normal form: this step leaves the program in a state where most variables are defined only once and a large number of copies exist in the program. This is clearly non-optimal because such quantity of copies is cumbersome to execute and a large number of virtual register can burden subsequent compiler modules, affecting compilation times. For these reasons we implemented a *Copy Coalescing* [13] algorithm, which scans the code for copies and tries to assign the same name to both the source and the destination variables involved in the copy. This is safe if the variables involved have live ranges that do not overlap. Beside optimizations based on dataflow analyses, the optimizer also provides algorithms for simplifying the structure of the control flow graph, 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 basic blocks [12].

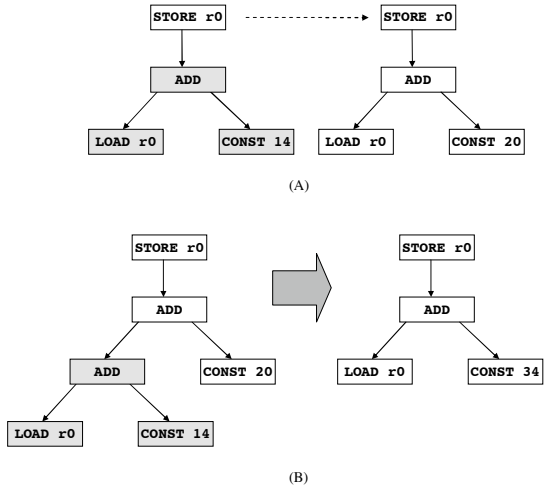

Figure 4. Optimization of packet demultiplexing code

Although the architecture-independent optimization algorithms implemented look simple and are widely known from classical compiler theory, they have proven to be extremely effective for two main reasons: (i) packet-processing applications use 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 a specific back-end, as will be detailed in Section 6. The combination of both architecture-independent and target-specific optimizations results in the production of code that in some cases is faster than the one generated by a commercial C compiler, as shown in Section 7.

6. COMPILER BACK-ENDS

The back-end for the Xelerated X11 network processor has been presented in [14]; hence this section will focus on the back-ends for the Intel x86 and Cavium Octeon platforms.

6.1 X86 Back-end

The x86 back-end follows the Just-In-Time paradigm: for each NetPE composing a NetVM application it generates the binary code for a function receiving an Exchange Buffer as an argument. While the process of translating the intermediate representation to machine code is quite similar to classical compilation, the NetVM model opens some new possibilities for low-level optimization.

First the back-end translates the tree-based intermediate representation generated by the upper layers of the compiler into the LIR. This task is handled through a Bottom-Up Rewriting System (BURS) [15], which executes a tree-matching algorithm driven by architecture-specific rules that specify how a portion of the intermediate representation (i.e. an expression sub-tree) should be translated into target instructions. In particular, different rules can relate to overlapping tree patterns, and the BURS is able to chose the best (i.e. the less expensive) combination that covers the most extended expression tree. BURS can be configured to recognize very specific patterns that can be part of an algorithm, enabling a very flexible approach in the

creation of the target code. For instance, an algorithm made up of three pieces *ABC* can be implemented as *AB* in software and *C* in hardware on one platform, and as *A* in software and *BC* in hardware on another platform.

The second step is the register allocation, whose task is to assign a machine register or a memory location to a live range of the program. We implemented a classical global register allocation algorithm through graph coloring [16][17], using the spill heuristic proposed in [18] for minimizing spill costs and for guaranteeing an optimal utilization of machine registers.

6.1.1 Intel X86 low-level optimizations

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

With respect to the first kind of optimization, the CISC-based Intel x86 includes powerful and complex instructions, which allow specific NetIL patterns to be translated into a single x86 instruction, with the result of minimizing the code size. The BURS instruction selection algorithm makes this operation straightforward. For example, Figure 5 presents an x86 code fragment that calculates the length of the IP options fields with both its naïve and its optimized version. Since 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 through the x86 `LEA` (Load Effective Address)[1] instruction, which exploits the Memory Management Unit of the processor.

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]`

Figure 5. Exploiting the Intel x86 instruction set

On the other hand we implemented special rules for optimizing frequent operations of packet-processing applications. For example, these often need to load a field from the packet header, perform some calculation and compare it with a constant value. However packets contain data organized in network byte order, which is big-endian, while x86 uses the little-endian convention. This requires swapping the data contained in the packet buffer before starting the processing. Our solution, instead, uses the BURS to recognize those patterns of instructions and move the byte swapping operation to compile time. In other words, whenever possible, instead of generating code for swapping the bytes of a register at runtime, the compiler swaps the constant during the compilation, thus producing more efficient code. A simple example of the use of this technique is presented in Figure 6, which refers to the control that determines if an Ethernet header is followed by an IP header.

[1] 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.

Non optimized	Optimized

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

Figure 6. Constant byte order swapping optimization

Another common operation in packet-processing applications is represented by the multi-way branch, modeled after the switch-case construct of the C language. The back-end includes a switch lowering module that follows an approach similar to the one implemented in the LLVM compiler [19], which is able to select the best mapping algorithm, according to the cardinality and the density of the case set.

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.

6.2 Octeon Back-end

Beside the x86 back-end, we implemented a back-end targeting the Cavium Octeon network processor. We will first present a short description of the characteristics of the processor before introducing how the NetVM model is mapped on it.

6.2.1 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 600 MHz. Each core has a private L1 cache, while the L2 cache and DRAM are shared. Although the L2 cache and DRAM are physically shared, the cores cannot communicate through the memory because of their private virtual memory space. Communication primitives between cores are provided by specific hardware mechanisms. 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 *Packet Order Work* (POW). Works have many attributes that determine how the POW schedules them to the cores. For example the programmer can specify different QoS levels associated with different kinds of traffic, since the unit receiving incoming packets can parse the packet header, providing a preliminary classification. The most important attribute is the *group*: in fact cores subscribe to groups and the POW schedules works to the cores according to the subscribed groups. When a core terminates its job, it can submit the work to another group, i.e. to another core, or send the packet out to a network interface.

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

6.2.2 The compiler back-end for the Cavium Octeon

In this case the compiler follows the Ahead-Of-Time model, which produces as its output several assembly files, C listings and configuration files. The result is a native application running on the NP hardware with a minimal runtime environment, as the processor units are exploited to implement natively the NetVM

model. In fact, the code generation is not different from the x86 back-end (i.e. it implements the BURS instruction selection and global register allocation), while the mapping of native hardware functionalities deserves some more discussion and represents the most valuable part of this work. Particularly, this includes the mapping of the Exchange Buffer (i.e., the memory that contains the packet) on native hardware structures, and the mapping of 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 POW 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. Currently, our prototype exploits only one core, hence it implements dynamic NetPE linking as in the x86 back-end and exploits the POW unit only for receiving and transmitting packets from the external world. However the general mechanism is already in place and can be used as a starting point for 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, which must be initialized before starting the program, 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 [20], 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, which has to be 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 the graph in the Low Latency memory to be used. The hardware unit automatically loads data from the packet memory and uses it to traverse the graph in the LLM, while searching for a match.

Finally, the runtime environment for this back-end is very simple and it consists of an initialization routine (automatically emitted by the compiler) that initializes the processor units and instantiates the memory structure needed by the NetVM instance. The only task of the runtime environment is then to receive packets from interfaces and to pass them to the NetVM.

7. EXPERIMENTAL EVALUATION

This section presents some tests that demonstrate the performance of the NetVM model and of its compiling infrastructure compared to other technologies. The tests are based on two applications written for the NetVM: *PFLCompiler* [11], a compiler for automatically generating packet-filtering programs starting from a filtering expression and a database containing protocol descriptions in terms of field format and encapsulation conditions, and *NetVMSnort* [21], a port of the popular open-source Intrusion Detection System *Snort*.

7.1 Testing the x86 back-end

Tests on the x86 platform measure the performance of the code emitted by our compiler compared to two other targets. The first one is the code generated by the BPF virtual machine, which is able to generate native assembly through the WinPcap Just-In-Time compiler. Although the WinPcap JIT compiler is very simple compared to our compiling infrastructure, it provides a useful benchmark with a well-known and widely-used architecture. The second target is made up of a set of native programs created in C language and compiled with Microsoft Visual Studio, which represent the real touchstone of our solution. The native C filters use a custom macro to speed up byte-ordering operations, instead of using the standard `ntoh()` functions of the C standard library.

We defined five packet filters[2] with different complexity, and we profiled the execution time through the RDTSC assembly instruction available on the x86 architecture. Tests were performed on a Windows-based machine, equipped with a Pentium 4 processor, running at 3GHz with Hyper-Threading and 4GB of memory.

Results presented in Table I show that our compiler generates code that is faster than that produced by the other technologies under testing. Main reasons rely on the intrinsic properties of the NetVM model, which exports some useful information to the compiling infrastructure, thus enabling very effective, albeit simple, optimizations (such as compile-time constant swapping). Since the characteristics of packet-processing applications are taken into consideration in the entire compilation process, the NetVM compiler can perform more aggressive optimizations than its counterparts. Notably, this is obtained with a limited set of optimizations compared to commercial compilers (such as Microsoft Visual Studio). Additionally, results show that both the mid-level optimizations and those implemented in the x86 back-end introduce a substantial boost in performance (third column) compared to non-optimized code (second column). These figures represent an improvement of the results presented in [11] in which no back-end optimizations were used.

Table I. Filtering time on the x86 back-end (ticks)

Filter	NetVM no opt	NetVM opt	BPF	Native
1	23	7	36	8
2	26	12	39	26
3	30	15	39	13
4	52	39	76	61
5	35	21	43	34

[2] 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 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.

7.2 Testing the Octeon back-end

The first test on the Octeon back-end shows the results obtained with the same five filters already presented in the previous section. Due to the lack of a BPF JIT compiler for this platform, NetVM filters are compared only to handwritten ones, the latter using the GNU C compiler (GCC). Results (in clock ticks) are presented in Table II.

Also in this case the code generated through the NetVM compiler is more efficient than that produced by the counterpart, thanks to the set of optimizations performed before emitting the code. In this case, the number of ticks is a good indication of the number of instructions emitted for each filter, because the Octeon processor is based on a MIPS pipelined architecture where most instructions are executed in exactly one clock cycle. These numbers can be further improved (although this is left to future work) by integrating a proper instruction reordering phase to avoid pipeline stalls.

Table II. Filtering time on the Octeon back-end (ticks)

Filter	NetVM	Native
1	9	8
2	14	15
3	17	20
4	51	62
5	29	32

On the Octeon platform we also executed the NetVMSnort application. Although a direct comparison with the original Snort is not possible (processing algorithms are not exactly the same, and the original Snort does not run on the Octeon platform because of memory limitations), the main result is that NetVMSnort compiles and runs on the Octeon platform and is able to exploit native hardware coprocessors. This demonstrates the possibility of mapping even a complex NetVM application on this architecture, hence the validity of the NetVM model. Furthermore, Table III shows the comparison between the time spent in coprocessors (out of the total time used by the application to complete its job) between the x86 platform, where string-matching is executed in software, and the Octeon platform, where string-matching is executed through a hardware DFA engine, demonstrates that the NetVM model enables the efficient exploitation of native hardware features on platforms in which these are available.

Table III. String matching performance on Octeon and x86

Platform	Percentage of the time spent in string matching
Octeon	3.79%
x86	13.44%

8. CONCLUSIONS

This paper presents the design and implementation of an optimizing multi-target compiler and run-time system for the NetVM model.

The aim of this work is to demonstrate that the virtual machine paradigm is applicable to packet-processing applications without affecting their performance, and greatly improves their portability. Our compiler allows the execution of NetIL code on different architectures, without performance penalties compared to hand-written code. We also show that the NetVM model is effective in exploiting the hardware features available on real network processors.

Although we have obtained excellent results, the implementation can still be further improved. A direction for future work is the study of specific medium-level optimizations for packet-processing applications. Another topic that we currently do not take into account is exploiting multiprocessor capabilities of NPs. In fact we can easily map each NetPE on a different core, but in the future we aim to find an automatic mechanism for splitting a generic application on multiple cores.

9. ACKNOWLEDGEMENTS

The authors wish to thank Marco Bergero and Pierluigi Rolando for the contribution they have given respectively in the development of the NetVM runtime environment and of the optimization framework presented in this work.

10. REFERENCES

[1] M. Baldi and F. Risso. Towards effective portability of packet handling applications across heterogeneous hardware platforms. In *IWAN 2005: Proceedings of the 7th Annual International Working Conference on Active and Programmable Networks*, Sophia Antipolis, France, November 2005.

[2] M. Baldi and F. Risso. A framework for rapid development and portable execution of packet-handling applications. In *ISSPIT 2005: Proceedings of the 5th IEEE International Symposium on Signal Processing and Information Technology*, Athens, Greece, December 2005.

[3] Cavium. Networks. Octeon Network Processors. http://www.caviumnetworks.com

[4] Xelerated. Xelerator X11 network processor. http://www.xelerated.com

[5] 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.

[6] R. Ennals, R. Sharp, and A. Mycroft. Linear types for packet processing. In *ESOP 2004: Proceedings of the 13th European Symposium on Programming*, pages 204–218, Barcelona, Spain, March 2004.

[7] R. Ennals, R. Sharp, and A. Mycroft. Task Partitioning for Multi-core Network Processors. In *Compiler Construction*, volume 3443/2005 of *Lecture Notes in Computer Science*, pages 76–90. Springer Berlin/Heidelberg, March 2005.

[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 *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 224–236, New York, NY, USA, 2005. ACM.

[9] G. Memik and W. Mangione-Smith. Nepal: A framework for efficiently structuring applications for network processors. In *Proceedings of the Network Processor Workshop in conjunction with 9th International Symposium on High Performance Computer Architecture (HPCA-9)*, Anaheim, California, February 2003.

[10] 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.

[11] O. Morandi, F. Risso, M. Baldi, and A. Baldini. Enabling flexible packet filtering through dynamic code generation. In *ICC 2008: Proceedings of the IEEE International Conference on Communications*, Beijing, China, May 2008.

[12] S. S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[13] 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.

[14] O. Morandi, F. Risso, P. Rolando, O. Hagsand, and P. Ekdahl. Mapping Packet Processing Applications on a Systolic Array Network Processor. *In HPSR 2008: Proceedings of the IEEE 2008 International Conference on High Performance Switching and Routing*, Shanghai, China, May 2008

[15] 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.

[16] L. George and A. W. Appel. Iterated register coalescing. *ACM Trans. Program. Lang. Syst.*, 18(3):300–324, 1996.

[17] P. Briggs, K. D. Cooper, and L. Torczon. Improvements to graph coloring register allocation. *ACM Trans. Program. Lang. Syst.*, 16(3):428–455, 1994.

[18] 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.

[19] A. Korobeynikov. Improving switch lowering for the llvm compiler system. In *SYRCoSE 2007: Proceedings of the 2007 Spring Young Researchers Colloquium on Software Engineering*, Moscow, Russia, May 2007.

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

[21] O. Morandi, P. Monclus, G. Moscardi, and F. Risso. An intrusion detection sensor for the netvm virtual processor. Technical report, Politecnico di Torino, September 2007. TR-DAUIN-NG-02