The Case for an Intermediate Representation for Programmable Data Planes

Muhammad Shahbaz and Nick Feamster
Princeton University

Abstract

Software-Defined Networking (SDN) switch vendors are interested in extending switch data planes to support new and continuously evolving network protocols (*e.g.*, NVGRE, VXLAN). Numerous commercial programmable data plane devices already enable a programmer to specify various aspects of the data plane including packet parsing, actions, and the layout of packet processing on the hardware device itself. Unlike OpenFlow-based devices, which only expose a series of *fixed* match-action table (MAT) abstraction, these specialized devices provide a more flexible abstraction for packet processing. Despite the increased programmability that these devices offer, however, the architecture of the target restricts the features that can be exposed to the programmer. Similarly, existing languages for programming the data planes in such devices (*e.g.*, P4) assume a specific computational model, resembling the architecture of the device for which they are targeted for. Unfortunately, this model leads to similar limitations as in OpenFlow, where the high-level specification is coupled to the underlying device model.

In this paper, we introduce *NetASM*, an *intermediate representation* for programmable data planes. *NetASM* is a device-independent language that is expressive enough to act as the target language for compilers for high-level languages, yet low-level enough to be efficiently assembled on various device architectures. It enables conventional compiler optimization techniques to significantly improve the performance and resource utilization of custom packet-processing pipelines on a variety of targets.

Categories and Subject Descriptors: C.2.1 [Computer-Communication Networks] *Network Architecture and Design*

General Terms: Algorithms; Design; Experimentation

Keywords: Software-Defined Networking (SDN); NetASM; Intermediate Representation (IR); Programmable Data Planes

1 Introduction

Emerging chipsets and platforms that support programmable data planes (*e.g.*, RMT, Doppler, Corsa, and Flexpipe) enable increasingly more fine-grained, flexible control over how the network device processes packets. These platforms offer for defining custom packet formats through specialized parsers, as well as ways to reconfigure the processing pipeline itself (*e.g.*, altering the number of stages, adding state to various stages in the pipeline).

SOSR2015, June 17–18, 2015, Santa Clara, CA, USA
© 2015 ACM. ISBN 978-1-4503-3451-8/15/06…$15.00
DOI: `http://dx.doi.org/10.1145/2774993.2775000`

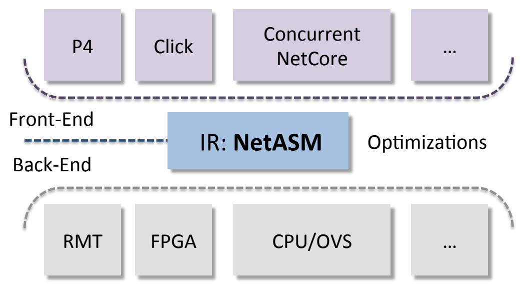

Figure 1: NetASM *is an intermediate representation that compilers can use to optimize packet processing pipelines from a variety of high-level languages to a diversity of targets.*

Yet, these devices remain difficult to program and configure, and there is currently no good way to write a program that specifies the configuration and layout of one of these custom packet processing devices. Ultimately, as occurred in conventional software engineering, we expect that network programmers will organize their programs into reusable libraries and composable code so that they can reuse these libraries to write more complex packet processing programs. For example, one operator might write logic for doing packet parsing, a second might write logic for an ACL and, similarly, a third might write logic for an IPv4 router. These packet processing modules will likely be reusable; they may also be in entirely different high-level languages. As programmable devices proliferate—each with potentially different high-level languages for configuring them—programmers will need mechanisms for reusing and composing these modules for a single hardware target.

Unfortunately, the use of multiple languages tends to make it more challenging to achieve direct compilation to a hardware target. For example, faced with a proliferation of high-level programming languages (*e.g.*, C, C#, Java, Python) compiler designers were faced with the issue of how to compile these languages to different target architectures (*e.g.*, AMD, ARM, and Intel's x86). Thus, instead of compiling each language directly to a given target, designers developed an *intermediate representation*. The intermediate language acted as a sweet-spot in dividing the compiler tasks into two phases: (1) front-end and (2) back-end. The details of the high-level language were confined to the front-end, and the details of the target machine to the back-end.

In this paper, we introduce *NetASM*, an *intermediate representation* for network compilers that serves as the "narrow waist" between the high-level languages that are beginning to emerge (*e.g.*, P4) and rapidly proliferating set of target hardware platforms, as shown in Figure 1. The intermediate representation (IR) must be language- and target-independent, it must be expressive enough to be produced by language-specific front-ends, and it must be functional enough to produce layouts for a diverse set of hardware targets. It must also be structured in a way that allows for optimizing packet-processing

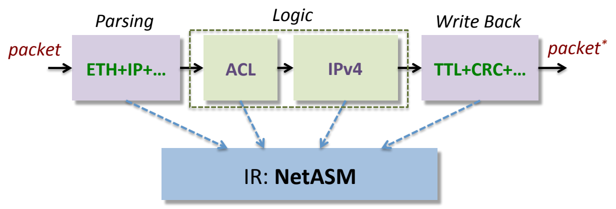

Figure 2: *Compiling applications written in (potentially different) high-level languages to* NetASM.

pipelines for area, power, and latency, using both target-specific and target-agnostic optimization techniques. Figure 2 shows how applications written in high-level languages can be compiled down into a single *NetASM* IR.

A significant advantage to a well-designed IR is the ability to optimize the layout of the resulting packet-processing program. Compiling programs that rely on composing reusable components and libraries can result in inefficiencies, mainly resulting from redundant operations. These redundancies often arise from either "dead code" or as a side-effect of writing the program in a high-level language that hides some of the low-level operations and the programmer has no choice but to use the higher-level constructs. For example, in P4, all processing relies on match-action tables, so a programmer must use the MAT construct for performing an operation as simple as loading a header field from memory. Programmers are not aware of these low-level features and cannot eliminate the redundancies themselves. In conventional software programs, these can lead to high memory and register usage, and more CPU cycles. For the case of programmable network data-planes, inefficiencies may result in longer packet processing pipelines (and hence higher latency), larger area on the hardware target, or more power consumption.

This paper presents the case for an intermediate representation for compiling high-level packet processing description languages (*e.g.*, Click, P4) to lower-level hardware targets. In presenting the need for *NetASM*, we address two challenging questions. First, the language must be both expressive enough to represent functions from a variety of languages that a compiler can use to produce output for multiple low-level data-plane targets. Section 2 describes the features of the language in detail. Second, it should be structured in a way to take advantage of existing frameworks for optimizing compilers, such as data- and control-flow analysis [7]. In Section 3, we explain how various optimizations both use and adapt existing optimizations and data-flow analysis techniques to optimize a high-level packet-processing program for area or latency; Section 4 evaluates these optimizations. Section 5 discusses related work, and Section 6 concludes with a summary and discussion of open research problems.

2 NetASM: An Intermediate Representation

NetASM instructions operate on state and produce a (modified) state. These operations can either operate on per-packet state (*i.e.*, state that travels with the packet in the pipeline) or on persistent state (*i.e.*, a table that is accessible from any stage in the pipeline). Persistent state is common in many abstract machine models; we introduce a new concept for an abstract machine model—per-packet state—to enable pipelined parallelism (Section 2.1). Within this pipelined execution model, we also enable both sequential and concurrent execution, as well as a new construct that makes it possible for arbitrary sequences of pipeline stages to operate atomically on persistent state (Section 2.2). In the sections below, we elaborate on these new features and why they are useful for programmable network data planes. Finally, we present the instruction set that we have designed to operate on state and perform these execution modes (Section 2.3).

2.1 Persistent State and Per-Packet State

NetASM adopts a computational model that resembles a register machine [5], yet we adapt this conventional model in several important ways. State in conventional register machines is always *persistent*. Content stored in the memory or registers will remain unchanged and will be available at each program point unless modified by some instruction in the program. In personal computer (PC) systems, this model reflects exactly what is happening; data is stored in RAM or registers and will only be modified by an instruction at some point in the program; otherwise, it will maintain its state for the entire execution of the program. If an instruction stores some results in a persistent state then that will immediately become available to all the packets in the pipeline, unless the state is overwritten by some other instruction in the program.

In contrast, in a network device, along with the *persistent state* that is maintained across packets, there is also *per-packet state* that is maintained as a packet traverses the device's pipeline. In a fully-pipelined program, each instruction will operate on a different packet and set of header fields (or *header set*); at each stage, results are written back to the packet and header fields, and then forwarded to the next instruction in the program. Persistent and per-packet state each provide benefits: persistent state makes it possible to maintain state across multiple packets (*e.g.*, for a stateful firewall or load balancer), and per-packet state allows us to speed up execution by running each stage of a packet processing pipeline in parallel.

To capture these two types of state behaviors, *NetASM* provides instructions that operate on both persistent state and per-packet state. Providing per-packet state as a native feature in the intermediate representation allows it to express highly efficient pipelined data planes, since different stages of the packet-processing pipeline can operate on different packets in parallel. (The *NetASM* model differs from conventional modes of state in other parallel systems, where cores are operating in parallel but not as part of a pipeline where packets with state are traversing.)

Instructions operating entirely on per-packet state in a *NetASM* program can be executed in parallel without any side-effects. Operations on persistent states, however, require careful analysis (*e.g.*, data dependence analysis) of the program; parallelism is inferred by adding redundant copies of the state element or by enabling control to enforce atomic access to the state element. Conventional abstract machine models for CPUs and even NPUs do not allow for this kind of pipelining as they only operate on persistent state and have a (default) sequential processing model, unlike the pipeline processing model that *NetASM* provides.

2.2 Sequential and Concurrent Execution

NetASM instructions can be executed either sequentially or concurrently. In sequential execution, each instruction in a *NetASM* program is executed one after the other. Because packets may be in different parts of the pipeline, these instructions may sometimes be able to execute in parallel. If, for example, these instructions operate on per-packet state, then it is possible to execute packets at different stages of the pipeline in parallel. If, on the other hand, these instructions access the same persistent state, the pipelined operations must be divided into atomic modules, each of which accesses the persistent state at different times.

Three *NetASM* instructions capture the necessary operations to support the different types of execution on different types of network

Type	Instruction
header	ADD, RMV
state	LD, ST, PUSH, POP
table	LDt, STt, LKt, INCt
control-flow	BR, JMP, LBL
arithmetic	OP
group	CNC, SEQ, ATM
special	CTR, DRP, CRC, HSH
other	ID, HLT

Table 1: *Entire* NetASM *instruction set.*

state: SEQ executes a block of instructions in sequence; packets move through a pipelined set of instructions. Multiple packets may be in the pipeline at any time, but each packet moves through the instruction sequence one after the other. A CNC block executes all instructions in a code block in parallel, and each instruction receives a pointer to the persistent state. If instructions try to access the same persistent state and are not in an ATM block will create a conflict. An ATM block indicates that a set of instructions should execute atomically, and thus any instructions within the block can share persistent state. By default, *NetASM* assumes a sequential execution model.

These three execution modes offer different benefits. Atomic operations (ATM) make it possible to write modular code blocks comprising multiple pipeline stages for target devices. It also allows instructions to share persistent state, thus enabling stateful applications (*e.g.*, stateful firewall). Similarly, these modes enable certain optimizations for area, latency or power by offering different ways to arrange various parts of the pipeline.

2.3 NetASM Instruction Set

The *NetASM* instruction set has only 23 instructions, as shown in Table 1. A *NetASM* program is a list of finite set of instructions. Using the *NetASM* primitives, a programmer can specify any kind of data plane: a conventional register machine model is Turing complete, and *NetASM*'s abstract machine model is an extension of this basic register model. *NetASM* provides instructions for the following operations: (1) load, (2) store, (3) computation, (4) branch, (5) header (*i.e.*, adding or removing headers in the header set), and (6) special operations (*e.g.*, checksum, hash).

3 Optimizations

Compilers often optimize programs expressed in intermediate formats by rearranging instructions; by modeling *NetASM* after conventional intermediate representations, we can take advantage of some of the same types of optimizations that conventional compilers use. Dead-code elimination, dead-store elimination, and code motion are examples of such function-preserving (and semantics-preserving) optimizations that compilers commonly apply. We first briefly describe the data-flow analysis framework used in building these optimizations. We will then consider optimizations that are inspired by conventional compilers and explain how we have adapted them to the setting of programmable network data planes.

Throughout the rest of the paper—both in presenting examples of optimizations and in evaluating them—we use the ACL-IPv4 benchmark. The logic for ACL and IPv4 is ported from the TTP specification, hosted at the ONF organization on Github [13]. The TTP specification only describes the match-action table (MAT) part of the pipeline, so therefore we had to write the Ethernet, IPv4 and TCP/UDP parsing and write-back logic. The final program is the composition of parsing, ACL-IPv4, and write-back logic. The *NetASM* source code for this example is available on Github [1]. We selected a real example from the ONF repository to emphasize that opportunities for optimizations regularly arise in real-world network packet-processing programs.

3.1 Data-Flow Analysis Framework

Data-flow analysis gathers information about a program to assist with optimization; it refers to techniques that derive information about the flow of data along program execution paths. The results of data-flow analyses all have the same form. For each instruction in the program they specify some property that must hold each time that instruction is executed.

We implemented two new kinds of data-flow analysis techniques, called *field-reachability* and *field-usability* analysis, for doing code-motion and dead-store elimination transformations. *NetASM* uses conventional optimization techniques in various ways: liveness analysis for dead-code elimination, reachability and reaching-definitions analysis for code motion, and usability analysis for dead-store elimination.

In field-reachability analysis, the compiler determines which fields are reachable at a program point p through some path prior to that point. In other words, we say that field f reaches the program point p if there is a path immediately following the instruction where the field f was last used. For example, if we look at the following code listing:

```
150 OP tcp_offset, tcp_offset, Add, 16
151 LD tcp_dst (tcp_offset, 16)
152 RMV tcp_offset
153 LD has_tcp, 1
```

`tcp_offset` is reachable at the entry of RMV instruction, at line 152, but not at the entrance of the LD instruction, at line 153.

The goal of field-usability analysis is to determine whether given field f created at point p is later used anywhere along the program path from p. For example, looking back at the code listing, above, we can see that field `tcp_offset` is not used after the RMV instruction, at line 152, but is used at the exit of the LD instruction at line 151. The field-usability analysis gathers these facts about the program which can then be used in removing dead fields at certain points in the program.

The liveness and reaching-definitions analysis are one of the most commonly used data-flow analysis with applications for various optimizations like dead-code elimination and constant propagation. *NetASM* implements these analyses in same way as in conventional compilers [7].

3.2 Dead-Code Elimination

There are various instances when an instruction can be considered "dead". In this paper, we apply the *NetASM* instruction set to show several examples where an instruction is declared dead can thus be removed from the program, thus reducing the overall area and latency.

One case where dead code arises is the presence of dangling fields (*i.e.*, fields that have been added in the instruction set but have never been used anywhere inside the code). Although clearly a programmer would never deliberately add fields to leave them unused, other optimizations can lead to such situations where the instruction previously using a field f is declared dead. To find dead fields in the program, we apply field-usability analysis.

Similarly, we can remove the accompanying RMV instruction from the code once the field f is no longer present in the header set.

To remove the RMV instruction the compiler must ensure that the field f that the RMV instruction is removing is not reachable from any path in the program prior to that RMV instruction.

Finally, instructions that assign some new value to a field, such as LD, OP, LDt, LKt, CRC, and HSH, can be declared dead if the field to which they assign some value to is not used anywhere later in the path. Standard liveness analysis can identify these cases. In the ACL-IPv4 example, the values of `eth_dst` and `eth_src` loaded at line 106 and 107, respectively, are never used throughout the remainder of the program and are overwritten again at line 274 and 275.

```
106   LD eth_dst, (0, 48)
107   LD eth_src, (48, 48)
```

Thus, the values loaded by the instructions at line 106 and 107, can be discarded from the program as dead-code.

3.3 Dead-Store Elimination

In NetASM, a programmer who adds a field to the header set need not manually remove it. A field will be valid and visible until the program explicitly issues an RMV instruction. Determining the optimal place to remove a field may be cumbersome for a programmer. This task becomes even more complex as programs get larger. (This is analogous to memory management in C.) Instead of asking the programmer to manage the removal of header fields from the header set, *NetASM* implements a new optimization, *dead-store elimination*. Using the field usability analysis, the compiler can determine when in the program a field is no longer used. At that point, the compiler adds a new RMV instruction to remove the field from the header set to ensure that the field is no longer visible to the rest of the program after that point. The code excerpt below from the same ACL-IPv4 example shows an opportunity for dead-store elimination.

```
150 OP tcp_offset, tcp_offset, Add, 16
151 LD tcp_dst (tcp_offset, 16)
152 LD has_tcp, 1
```

After line 151, the program no longer needs `tcp_offset` field and, thus, the field can be removed from the header set, as shown in the following example.

```
OP tcp_offset, tcp_offset, Add, 16
LD tcp_dst (tcp_offset, 16)
RMV tcp_offset
LD has_tcp, 1
```

3.4 Code Motion

Instructions might add a field to the header set long before the field is actually used. Doing so can create readable and manageable code; for example, one may want to add all fields of a header first before writing the rest of the program. A common example where this occurs is writing a parsing logic in the program: A programmer first adds fields to the header set and then loads them with the content of the packet (using the LD instruction).

As with inserting RMV instructions, it may be difficult for a programmer to manually determine an optimal place in the program to add a field to the header set. *NetASM* thus provides another optimization called *ADD code motion* that moves the ADD instructions closer to the point where they are first used. The compiler performs this operation using both field reachability and reaching-definitions analysis. Field reachability analysis indicates points in the program where a field was first used by an instruction. Using this information, the compiler moves the ADD instruction immediately before that point. This operation requires knowing the size of the field needed by the ADD instruction when adding the field in the header set; reaching-definition analysis provides this information. The compiler scans ADD instructions, and the compiler selects the one matching the given field. Consider a code excerpt from the same ACL-IPv4 example below:

```
47 ADD eth_dst, 48            106 LD eth_dst, (0, 48)
48 ADD eth_src, 48            107 LD eth_src, (48, 48)
49 ADD eth_type, 16           108 LD eth_type, (96, 16)
```

Lines 47–49 add Ethernet header fields `eth_dst`, `eth_src`, and `eth_type` to the header set, first. Later, lines 106–108 load values from the packet at given offsets into the header fields. In this case, all load operations will see the three header fields in their header set and thus incurring more cost. The compiler can reduce overall cost by moving the ADD instructions closer to the load operations where the header fields are first used, as shown below:

```
ADD eth_dst, 48
LD eth_dst, (0, 48)
ADD eth_src, 48
LD eth_src, (48, 48)
ADD eth_type, 16
LD eth_type, (96, 16)
```

In this case, the first LD instruction will now only have `eth_dst` in its header set.

RMV instructions that a programmer adds can be moved to earlier locations in the program (*i.e.*, just after the last use of the field). The compiler performs this operation using the same field usability analysis without including the RMV instructions, determining the last-used point for fields as the instruction just before the corresponding RMV instruction. Using this information, the compiler can insert new RMV instructions just after those instructions. We call this optimization *RMV code motion*.

4 Preliminary Evaluation

In this section, we perform a preliminary evaluation of the benefits of *NetASM*'s optimizations by evaluating them against our running ACL-IPv4 example. We first introduce and justify the abstract cost model that we use to evaluate the optimizations; we then compute the benefits of the optimizations using this cost model.

4.1 Cost Model

We evaluate *NetASM* based on an abstract cost model that we develop; we use this model to calculate the abstract area and latency of a *NetASM* program. The values of area and latency simply have weights that reflect the relative costs of each instruction; higher area and latency costs reflect more area usage and higher latency, respectively.

Area. The area cost (A_i) for each instruction (i) is a function of the sum of the size of fields (f) in its header set (H_i), its type, and the operation it performs. To model the effect of instruction's type and the operation it performs, we assign each instruction a weighing factor (w_i). Following equation shows how w_i is calculated for *NetASM* instructions.

$$w_i = \begin{cases} 4 & : (i = \texttt{STt} \mid \texttt{LKt})\ \& \\ & \quad (i.\texttt{table.type} = \texttt{CAM}) \\ 3 & : (i = \texttt{LDt} \mid \texttt{STt} \mid \texttt{LKt} \mid \texttt{INCt})\ \& \\ & \quad (i.\texttt{table.type} = \texttt{RAM}) \\ 2 & : (i = \texttt{LD}\ \&\ i.\text{source} = \text{Location}) \mid \\ & \quad (i = \texttt{OP}\ \&\ i.\text{operator} = (\text{Mul}|\text{Div})) \\ 1 & : \textit{otherwise} \end{cases}$$

For example, in the case of load (LD) instruction, the cost of the instruction is doubled if the source operand is of type Location (*i.e.*,

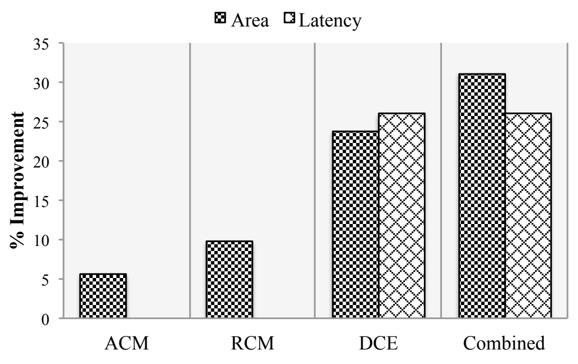

Figure 3: *Improvements in area and latency of the ACL-IPv4 application when different combinations of optimizations are applied.*

we are loading the value from a packet), so we assign a weighing factor (w_i) of 2. Similarly, for the OP instruction, w_i is 2 for multiplications and divisions. We assign table instructions weight 3 or 4 depending on whether they are operating on RAM or CAM tables, respectively. For all other cases w_i is 1. Thus, cost of area for an instruction is: $A_i = w_i \cdot (\sum_{f \in H_i} sizeof(f))$, where $sizeof$ returns the size of field (f) in bits. The cost of group instructions (*i.e.*, SEQ, CNC, and ATM) is the sum of the cost of the input code provided with that group instruction.

The cost of a table (t) is a function of the product of its size (s) and width, table type, and the field's match type. The width of a table is the sum of the size of fields in its field set (F_t). To take into account the effect of table type and field's match type on the actual cost, we assign a weighing factor (w_t) with each table,

$$w_t = \begin{cases} 2 & : t.\text{type} = \texttt{CAM} \\ 1 & : t.\text{type} = \texttt{RAM} \end{cases}$$

and (w_f) with each field in the table.

$$w_f = \begin{cases} 3 & : f.\text{match\_type} = \texttt{Ternary} \\ 2 & : f.\text{match\_type} = \texttt{Binary} \\ 1 & : \textit{otherwise} \end{cases}$$

Thus, we compute the area cost of a table as: $A_t = s \cdot w_t \cdot (\sum_{f \in F_t} (sizeof(f) \cdot w_f))$. The total area is the sum of all instructions and tables present in the program: $A = (\sum_{i \in I} A_i + \sum_{t \in T} A_t)$.

Latency. The cost of latency depends on the complexity of the operation that an instruction performs and is given by the following formula: $L_i = w_i$. The weighing factor indicates the latency that an instruction will have. For example, consider again the load (LD) instruction; if the source operand is Location, then its latency is 2; otherwise, it will be 1. Similarly, if a table instruction is using CAM table, its latency is 4; otherwise, it will be 3. The total latency is the sum of all instructions present in the program *i.e.*, $L = \sum_{i \in I} L_i$.

The ADD, RMV, LBL, and HLT instructions do not add to the area or latency cost of any program. An optimizer can then use this model to optimize the program for either area, latency or both at an abstract (*i.e.*, target-independent) level.

4.2 How Well Do the Optimizations Work?

We evaluated the application against four different combinations of optimizations and measured the improvement in area and latency with respect to the baseline values of the original program. We also measured the time it took to complete the optimizations. We calculated area and latency using the cost model from Section 4.1. The baseline area and latency for the ACL-IPv4 program are 47,470 bits and 127 cycles, respectively.

	ACM	**RCM**	**DCE**	**Combined**
Time (sec)	0.29	0.24	0.32	0.64

Table 2: *Optimizations Timing.*

Figure 3 shows the improvements for the ACL-IPv4 example as a result of applying various compiler optimizations. The compiler clearly achieves the largest improvements when the ADD code-motion (ACM), RMV code-motion (RCM), and dead-code elimination (DCE) are applied together: There is a 40% improvement in the final area consumed by the application, and a 26% improvement in latency.

The ACM and RCM optimizations only affect the area, as these optimizations only try to eliminate redundant use of header fields, and do not actually remove any dead code. Whereas, the DCE optimization actually removes redundant code from the program, thus, giving improvements in both area and latency of the program. Optimizing the code using ACM, RCM, and DCE took only 0.64 seconds (Table 2).

5 Related Work

Chipsets and hardware that support flexible and programmable data planes have recently proliferated. For example, Intel's FlexPipe architecture [9] supports programmable packet parsing and operations on fields using MATs. The RMT [4] is an architecture for a flexible packet processor that has a multi-stage MAT pipeline. Corsa [6] and Algo-Logic [2] have announced fully programmable network devices using Field Programmable Gate Arrays (FPGAs). Each of these targets currently has its own mechanisms for specifying and programming the hardware data plane. *NetASM* aims to provide an intermediate representation for compilation to any of these targets.

The advent of programmable hardware has also brought new programming languages to the fore. P4 [3] is a declarative language for expressing how packets are processed in the data plane using a forwarding model consisting of packet parser and MAT stages. P4 allows configuration of chipsets with exclusively programmable MATs, but it is not sufficient for other architectures that do not conform to the match-action paradigm. POF [12] provides a generic flow instruction set, but it does not support execution modes needed to efficiently compile a program to targets with different architectures. CNC [10] is a typed language for specifying routing policies and control flow using MATs. SDNet [11] provides a programming language that defines how an FPGA-based switch should process packets. OF-PI proposes an SDN ecosystem and a language for a specific type of abstract forwarding model based on MATs [8]. Despite these representations and hardware architectures, no compiler exists that can take an arbitrary high-level language for programmable data planes and compile to an arbitrary target. Some of these languages will be more suitable for one particular architecture than another. For example, P4 uses an abstract forwarding model that is inspired by the RMT architecture and, thus, provides a relatively direct compilation path for RMT. In the case of other architectures like Corsa and Algo-Logic, compilation from P4 might not be straightforward.

6 Summary and Future Work

We presented the case for a new intermediate representation, *NetASM*, that enables a compiler to optimize a high-level packet processing program for a diversity of targets. *NetASM* divides data-plane functions into small primitive operations, enables high-level languages to provide more abstract data plane representations, and provides an efficient assembly of these primitives to different target devices. *NetASM* uses a target-independent machine model and cost semantics to optimize the program for metrics such as area and latency.

Our research agenda includes completing the language specification of *NetASM* and building a compiler framework using different packet-processing specification languages (*e.g.*, P4) and data-plane targets. We will explore various opportunities for optimizations that can be applied across different classes of network device architectures. This will lead to better architectural explorations and, we believe a time will come when programming in high-level languages will become a norm and the performance of a switch would not only be determined by its raw speed but also by how well compilers can exploit its features.

Acknowledgments

The authors wish to thank Jennifer Rexford, David Walker, and Cole Schlesinger for helpful early discussions; Srinivas Narayana, Jennifer Rexford, and the anonymous SOSR reviewers for their feedback on earlier versions of this paper. This work was supported by NSF Award CNS-1539920.

References

[1] NetASM: ACL-IPv4-Example. `github.com/NetASM/ACL-IPv4-Example`. (Cited on page 3.)

[2] High Performance GDN 100G Top-of-Rack (TOR) Switch for Datacenter. `www.algo-logic.com/gdn-100g-tor-switch`, 2014. (Cited on page 5.)

[3] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014. (Cited on page 5.)

[4] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*, pages 99–110. ACM, 2013. (Cited on page 5.)

[5] A. W. Burks, H. H. Goldstine, and J. Von Neumann. *Preliminary discussion of the logical design of an electronic computing instrument*. Springer, 1982. (Cited on page 2.)

[6] Corsa Technology: SDN Data Planes. `www.corsa.com`, 2014. (Cited on page 5.)

[7] M. Lam, R. Sethi, J. Ullman, and A. Aho. Compilers: Principles, techniques, and tools, 2006. (Cited on pages 2 and 3.)

[8] OF-PI: A Protocol Independent Layer. `www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/OF-PI__A_Protocol_Independent_Layer_for_OpenFlow_v1-1.pdf`, 2014. (Cited on page 5.)

[9] R. Ozdag. Intel® Ethernet Switch FM6000 Series-Software Defined Networking. *See goo.gl/AnvOvX*, 2012. (Cited on page 5.)

[10] C. Schlesinger, M. Greenberg, and D. Walker. Concurrent NetCore: From Policies to Pipelines. (Cited on page 5.)

[11] SDNet: Software Defined Specification Environment for Networking. `www.xilinx.com/applications/wired-communications/sdnet.html`, 2014. (Cited on page 5.)

[12] H. Song. Protocol-oblivious forwarding: Unleash the power of sdn through a future-proof forwarding plane. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 127–132. ACM, 2013. (Cited on page 5.)

[13] Table Type Patterns (TTP) Repository. `github.com/OpenNetworkingFoundation/TTP_Repository`. (Cited on page 3.)