NetVRM: Virtual Register Memory for Programmable Networks

Hang Zhu
Johns Hopkins University

Tao Wang
New York University

Yi Hong
Johns Hopkins University

Dan R. K. Ports
Microsoft Research

Anirudh Sivaraman
New York University

Xin Jin
Peking University

Abstract

Programmable networks are enabling a new class of applications that leverage the line-rate processing capability and on-chip register memory of the switch data plane. Yet the status quo is focused on developing approaches that share the register memory statically. We present NetVRM, a network management system that supports *dynamic* register memory sharing between multiple *concurrent* applications on a programmable network and is readily deployable on commodity programmable switches. NetVRM provides a *virtual register memory abstraction* that enables applications to share the register memory in the data plane, and abstracts away the underlying details. In principle, NetVRM supports *any* memory allocation algorithm given the *virtual register memory abstraction*. It also provides a default memory allocation algorithm that exploits the observation that applications have diminishing returns on additional memory. NetVRM provides an extension of P4, P4VRM, for developing applications with virtual register memory, and a compiler to generate data plane programs and control plane APIs. Testbed experiments show that NetVRM generalizes to a diverse variety of applications, and that its utility-based dynamic allocation policy outperforms static resource allocation. Specifically, it improves the mean satisfaction ratio (i.e., the fraction of a network application's lifetime that it meets its utility target) by 1.6–2.2× under a range of workloads.

1 Introduction

Programmable networks are a new paradigm that changes how we design, build and manage computer networks. Compared to traditional fixed-function switches, programmable switches allow developers to flexibly change how packets are processed in the switch data plane. The programming model of programmable switches are based on a multi-stage packet processing pipeline [8, 9].

Programmable switches provide different types of stateful objects that preserve states between packets, such as tables, counters, meters and registers. Among them, registers allow packets to read and write various states at line rate, which then affects how the following packets are processed. Such data-plane-accessible *register memory* is one of the defining features of programmable switches, and enables a new class of *reg-stateful* applications which utilize the on-chip register memory to realize various functionalities. These reg-stateful applications include not only the innovations in traditional network functions like congestion control [45], load balancing [25, 35] and network telemetry [1, 18], but also novel use cases beyond traditional networking, such as caching [23, 32], consensus [13, 14, 22] and machine learning [42, 43].

Given the rise of reg-stateful applications, an important open problem is how to support multiple concurent reg-stateful applications running efficiently on a programmable network [51]. The utility of reg-stateful applications is usually decided by the amount of allocated register memory and the real-time network traffic [18, 23, 34, 47, 54, 58]. Thus, it is essential to dynamically allocate the limited register memory between multiple applictions to optimize the multiplexing benefits. Yet existing approaches of running multiple concurrent applications on programmable networks allocate register memory statically [19, 44, 49, 56, 57]. Changing the amount of register memory for one application would require recompiling and reloading the switch program, which would disrupt the operation of the switch.

In this paper, we propose NetVRM, a network management system that supports *dynamic* register memory sharing between multiple *concurrent* applications on a programmable network. NetVRM advances the status quo with three major features: The first one is a novel *virtual register memory abstraction*, which allows the register memory in the switch data plane to be dynamically allocated between multiple concurrent applications *at runtime*, without recompiling and reloading the data plane program. The second one is a dynamic memory allocation algorithm, which efficiently arbitrates the memory usage between concurrent applications based on the real-time utility measurements. The third one is a language extension and a compiler to generate data plane programs with the virtual register memory abstraction and efficient C++ control plane APIs for high-speed virtual register memory configuration.

The virtualization of register memory allows its dynamic allocation. Our approach is inspired by traditional virtual memory designs in operating systems, but programmable switches introduce two new challenges. First, register memory is distributed over multiple pipeline stages, and each register can be accessed only from one stage. Second, switch applications can access register memory from both the data

plane and control plane. NetVRM's memory system design is tailored to these characteristics. It places a page table at the front of the virtual register memory's processing pipeline, using it for memory translation in the data plane. The page table indexes the register memory regions allocated to each application in every stage. The switch control plane manages memory allocation. NetVRM also mediates application accesses to register memory from the control plane to ensure addresses are correctly translated.

NetVRM's dynamic memory allocation policy exploits the fundamental tradeoff between memory consumption and application utility. In particular, it leverages *diminishing returns*: the observation that, for most reg-stateful applications, the benefit of additional memory decreases with the amount of allocated memory [18, 23, 34, 47, 58]. For example, after a certain point, NetCache [23] cannot further improve the throughput significantly. More importantly, the memory-utility relationship changes both in the *temporal* and *spatial* dimensions based on application characteristics and traffic conditions. For example, the amount of register memory needed by NetCache depends on the request pattern, which can change over time and even vary across different switches. We design an online algorithm that does global memory allocation between applications in the network to maximize multiplexing benefits.

To make it easy to develop applications with NetVRM, we propose P4VRM, an extension to P4 [8]. P4VRM allows developers to virtualize register memory with a few simple modifications to existing P4 code: they mark register arrays to be virtualized and add online utility measurement primitives provided by P4VRM. The compiler takes multiple P4VRM programs as input and outputs a single P4 program with the virtual register memory abstraction and all the applications' functionalities, and generates the control plane APIs for high-speed virtual memory configuration.

In summary, we make the following contributions.

- We propose NetVRM, a network management system that exposes a virtual register memory abstraction to enable dynamic register memory sharing between multiple concurrent applications on a programmable network at runtime without recompiling and reloading.
- We design a dynamic memory allocation algorithm to efficiently allocate register memory between applications to maximize multiplexing benefits.
- We propose P4VRM, a data plane program extension, and provide a compiler to easily equip the data plane programs with virtual register memory and generate control plane APIs for efficient virtual memory configurations.
- We implement a NetVRM prototype. Testbed experiments on a variety of applications show that compared to static memory allocation, NetVRM improves the mean satisfaction ratio (i.e., the fraction of a network application's lifetime that it meets its utility target) by 1.6–2.2× under a range of workloads.

2 Motivation and Related Work

2.1 The Case of Dynamic Register Memory Allocation

Concurrent reg-stateful network applications. There are two broad types of objects provided by commodity programmable switches on the data plane—*stateless* objects, such as metadata, packet headers, and *stateful* objects, such as match-action tables, counters, meters, registers. Among them, registers, as one of the defining features of new-generation programmable switches, provide data-plane-accessible *register memory* for packets to read and write various states at line rate and enable much of the latest exciting research [14, 22, 25, 35, 42, 43, 45]. Register memory is implemented with standard SRAM blocks and can be read and written by both the control plane and data plane. Stateful Arithmetic and Logic Unit (ALU) performs register memory access and modification by executing a short program that involves register data, metadata and constant. The register memory is usually organized as register arrays. Each register array consists of several register slots with the same width and can be addressed by index (direct mapping) and hash (hash mapping). We refer to the network applications that use the register memory as *reg-stateful* applications.

Besides the rise and evolution of reg-stateful applications, modern cloud service providers usually serve multiple tenants concurrently [6, 30]. They allow tenants to run different network applications dynamically. For example, Azure and AWS provide a variety of network applications [5, 7] to their tenants, such as network address translation (NAT), load balancer, and network monitoring. We anticipate that the reg-stateful applications will be provided to tenants as programmable switches are being integrated in cloud networks, including both the datacenter networks and the wide area networks that connect the datacenter networks.

Necessity and potential benefits of network-wide dynamic allocation. The register memory on programmable switches is fundamentally limited by the hardware. For example, the maximal size of register memory on each stage is only a few Mb on the Intel Tofino switch [50]. Besides the limited register memory, there is a fundamental trade-off between memory consumption and application utility (e.g., its performance or accuracy) in many reg-stateful network applications [18, 23, 34, 47, 58]. Although some applications have a fixed memory requirement, most can operate with different amounts of available memory. Notably, our key observation is that applications generally exhibit *diminishing returns* [18, 23, 34, 47, 58]. The utility improvement decreases with more memory, and for many applications, additional memory has *no* utility after a point. We demonstrate the diminishing returns for four applications in Appendix A, including heavy hitter detection (HH) [54], newly opened TCP connection detection (NO) [55], superspreader detection (SS) [54] and NetCache [23]. The utility is measured using memory hit ratio (§5.1).

In all cases, the amount of memory affects the application utility, and such effects depend heavily on the workload. For example, NetCache [23] needs different amount of register memory with different skewed workload to deliver the same utility (Appendix A). Without dynamic allocation, this presents a formidable deployment challenge because the workload can vary in both temporal and spatial dimensions: different storage clusters see radically different workloads, and even a single cluster's request pattern changes over time (e.g., on a diurnal cycle) [4].

The diminishing returns and the temporally and spatially dynamic workload together also provide the opportunity to maximize resource multiplexing benefits by efficiently arbitrating the memory usage between concurrent applications.

2.2 Target and Scope of NetVRM

Target applications. The reg-stateful applications that can benefit from NetVRM must have the following properties.

- **They are elastic** (§5). An inelastic application (e.g., NetChain [22]) that has fixed virtual memory requirement can be supported by NetVRM, but cannot benefit from dynamic memory allocation.
- **The data plane programs have to meet the constraints in P4VRM** (§6), such as stateful ALUs since each operation of one register array must be associated with a specific stateful ALU.
- **The application utility should be obtained instantaneously** (§5.1). It can be computed on the switch (e.g., hit ratio as the default utility) or reported by applications.

We remark that there are a wide range of applications with the above properties, such as measurement applications [18, 39, 47], applications with approximate data structures [20, 34, 54], and caching applications [23, 33].

Register memory as the scope. There are a variety of resource types on a programmable switch, such as register memory, SRAM used for tables, TCAM and action units [51]. NetVRM focuses on dynamic allocation for register memory for three reasons. First, we observe that many reg-stateful applications are bottlenecked by register memory. Second, dynamic allocation of other resource types (e.g., match-action tables, TCAM) has been well-studied in the context of Software-Defined Networking (SDN) with traditional fixed-function switches [17, 21, 36, 46]. Third, current switch hardware cannot dynamically reallocate other resource types without rebooting the entire switch [51]. NetVRM is readily deployable on existing programmable switches.

Switch memory available that can be used as virtual register memory could be limited because a certain amount of memory has to be set aside for basic networking functionality, such as L3 routing, and inelastic applications (see §5). The evaluation in §8 shows that NetVRM outperforms the alternatives, regardless of *how much* physical memory is available for virtualization and dynamic allocation. Thus, NetVRM continues to be effective even as the memory for basic networking functionality and inelastic applications grows in size, leaving behind less memory for dynamic allocation.

2.3 Existing Solutions and Limitations

Recently, several existing works have explored how to support multiple applications on a programmable switch [19, 44, 48, 49, 56, 57]. At a high level, these solutions fail to meet the requirement of dynamic register memory allocation because of at least one of three limitations as follows.

- **Static binding of register memory.** Some of the existing work combine or merge multiple applications into one monolithic data plane program [19, 48, 56, 57] in compilation time. And the binding between register memory allocation and applications is static. Changing the allocation requires the data plane program to be recompiled and reloaded, during which the switch has to be stopped and restarted. This interrupts the operation of all applications on the switch, even the basic ones such as L3 routing.
- **Lack of a real switch environment.** Most of the existing solutions ignore the practical hardware constraints and are not applicable on a real ASIC-based switch (e.g., Intel Tofino [50]). For example, P4VBox [44] provides parallel execution of virtual switch instances on NetFPGA. MTPSA [49] realizes a multi-tenant portable switch architecture on NetFPGA and BMv2, a reference P4 software switch [3]. HyPer4 [19] and HyperV [56] realize the virtualization on software switches (e.g., BMv2, DPDK).
- **Not doing network-wide dynamic allocation.** Network resource allocation has been well studied for SDN with traditional fixed-function switches [16, 17, 21, 36, 37, 46]. For example, DREAM [36] does dynamic allocation for TCAM between measurement applications. However, none of the existing work has disclosed the potential benefit of a network-wide dynamic allocation for the *register memory* on programmable networks.

There are other related works that have explored how to manage and improve network applications on programmable networks. TEA [27] provides external DRAM for storing *table entries*, not *register memory*. Dejavu [52] utilizes the multiple pipelines and resubmission to fit a service chaining in one single switch. RedPlane [28] enables fault-tolerant stateful applications by designing a practical, provably correct replication protocol. NetVRM targets *register memory* and provides a new system for sharing it between multiple concurrent *reg-stateful* applications dynamically.

3 NetVRM Overview

NetVRM is a network management system that supports dynamic register memory sharing between multiple concurrent applications on a programmable network. Figure 1 shows an overview of NetVRM. NetVRM includes three critical components: virtual register memory, dynamic memory allocation and the P4VRM compiler. It abstracts away the complexities of allocating physical memory in each application, increases

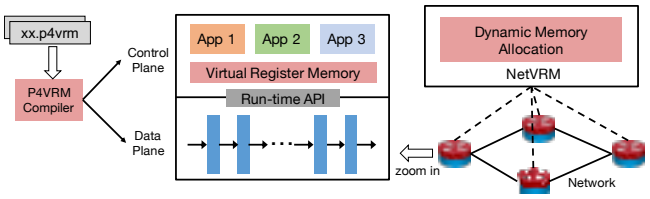


Figure 1: NetVRM overview.

memory utilization via statistical multiplexing, and provides P4VRM as an extension of P4 for developing applications with such virtual register memory.

Virtual register memory (§4). NetVRM exposes a virtual register memory abstraction to applications. The virtual register memory component in every switch hides the underlying details of the physical register memory that may span multiple stages and be shared with multiple applications. We design a custom data plane layout and an address translation mechanism to realize the virtual memory. The data plane layout composes the register arrays in multiple stages to one large register array, and allocates the large array to applications. Memory translation contains two page tables. One page table is in the data plane that translates the memory addresses computed from packet headers for memory access during packet processing, and the other is in the control plane for NetVRM to query and update the virtual memory of applications. The two tables are synchronized and managed by NetVRM.

Dynamic memory allocation (§5). In principle, NetVRM can support *any* memory allocation algorithm built on top of the virtual register memory. NetVRM also provides a default network-wide memory allocation algorithm for applications *without* knowing the utility functions. The algorithm exploits the diminishing returns between memory usage and application utility to maximize resource multiplexing benefits. We leverage the observation that many applications use the switch as a performance accelerator and deal with insufficient switch memory by having some kind of fallback path, either through the switch control plane or the servers [23, 29, 47]. As such, we cast the resource allocation problem as satisfying as many application's requirements as possible with respect to available memory size. This allows operators to specify application-specific utility metric and target for each application, avoiding the need to compare different utility functions across applications. NetVRM also provides a default, application-agnostic metric—the memory hit ratio—for applications that do not define their own.

Language extension and autogeneration (§6). NetVRM provides P4VRM, an extension to P4 [8] for developers to develop P4 programs with virtual register memory, and a P4VRM compiler to compose and compile individual P4VRM programs of different applications to one single P4 program with virtual register memory abstraction. The compiler also generates C++ APIs for efficient virtual register memory configuration in the control plane.

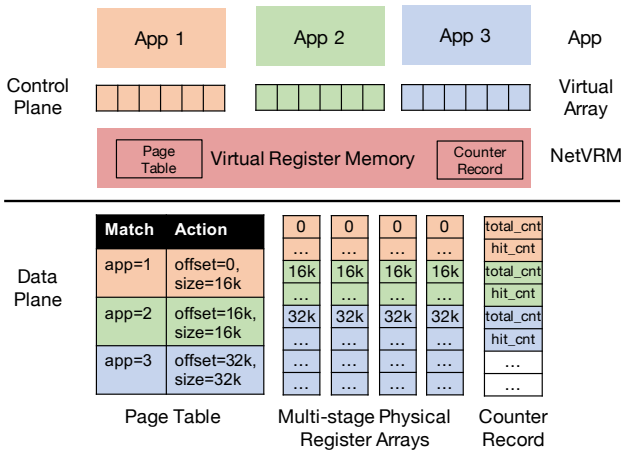


Figure 2: Virtual register memory design.

4 Virtual Register Memory

The register memory in the switch data plane is abstracted as register arrays for developers. The main problem of dynamically allocating memory is the coordination between multiple reg-stateful applications. Because register array definitions are hardwired in P4 programs, the code of an application has to be modified when other applications on the switch change, even if the application itself stays the same. NetVRM exposes a *virtual register memory space* to each application, which eliminates the coordination between applications. Each application is implemented with a virtual register array, without explicitly binding the register array to specific stages. As such, the application code does not need to be modified when the memory allocation changes. NetVRM is designed to manage the register memory and does not scarifice the support of recirculation.

Page table and counter record. A key challenge for memory virtualization on a switch, as opposed to a traditional CPU, is that the register memory can be accessed from both the data plane and the control plane (Figure 2). It is straightforward to implement the page table in the control plane. NetVRM simply does the translation in software. Specifically, it intercepts application memory accesses, uses the page table to perform the address translation, and then calls the memory access APIs of the switch driver to update the register memory configuration.

The page table in the data plane is more complicated, because it needs to be implemented using the programmable processing elements in the data plane. Figure 2 shows the design. The page table is implemented with a match-action table, and is placed at the stage before the physical register arrays to be virtualized. The match-action table matches on the application ID and identifies the location and size of the application's memory region (`offset` and `size`). These parameters are configured by the control plane at runtime as memory is allocated. We remark that *the page table does not introduce register memory overhead in common cases* (§7).

The counter record maintains two counters for each application, which only takes a small amount of memory. One is `total_cnt`, which tracks the total number of packets for an application. The other is `hit_cnt`, which tracks the number of packets that hit the switch register memory for each application. These counters are polled and reset periodically by the control plane to compute real-time memory hit ratios.

Memory layout. The memory layout partitions the physical register arrays *horizontally* across the stages. A virtual register array for an application is mapped to multiple blocks with the same start index (`offset` in the page table) and size (`size` in the page table) in each physical array. For example, in Figure 2 application 1 has a virtual array with 64K slots, which is mapped to $[0, 16K)$ in each physical array, and application 3 has a virtual array with 128K slots, which is mapped to $[32K, 64K)$ in each physical array.

This horizontal memory layout has three principal benefits. First, it decouples memory allocation from application code, and eliminates their static binding. The size of a virtual register array and its mapping to the physical arrays are represented by `offset` and `size` in action parameters, which can be dynamically changed at runtime, without recompiling and reloading the code in the data plane. Second, it enables fine-grained memory allocation. Because there are only a few stages (e.g., 10-20 stages) on commodity programmable switches [11, 50], our design can allocate the memory at row granularity (e.g., 8-slot granularity), which is fine-grained enough, compared with the total available slots on the switch (e.g., 512K). Third, it represents the memory layout using a small fixed-sized representation: only two variables (`offset` and `size`) per application. Although a more sophisticated memory layout might be able to achieve better space efficiency, more complex representations such as variable-length block lists would be challenging to implement efficiently in the data plane.

Address translation. Let the size of a virtual register array for an application be N. A virtual address $VA \in [0, N)$ is the index of the register slot in the virtual array. The physical address PA is computed by $PA = (VA/size,\ VA\%size + offset)$ after the page table, where $VA/size$ denotes the physical array index and $VA\%size + offset$ denotes the physical slot index in the corresponding stage. Division and modulo on arbitrary integers may not be supported in all switches. In such cases, we allocate virtual arrays with size to be a power of two, and implement these two operations with bit operations.

The above translation is sufficient for applications that directly access memory by VA. Besides these direct accesses, reg-stateful applications on programmable switches often use a lookup table or a hash function to access a register slot. Lookup tables use match-action tables to identify the address corresponding to a key (e.g., to find the memory location of an object in NetCache). We adapt the match-action table to hold a virtual address, then apply the VA to PA translation described above. Other applications use a hash function to map a subset of header fields to a register slot (e.g., hashing the source IP in heavy hitter detection). While in principle the same translation approach can be used, hardware constraints on the Tofino platform mean that hash functions need to be associated with a particular address range, and adding a variable offset to the output requires an additional stage. NetVRM uses a hash function h_size, selected during the page table lookup stage, which has output in $[0, size)$. Hash lookups first compute $h\_size(pkt.hdr)$, then, in a subsequent stage, translate that to the physical slot location: $PA = (h(pkt.hdr)\%k,\ h\_size(pkt.hdr) + offset)$, where k is the number of physical arrays.

Some applications may need large virtual slots, each of which may be larger than a physical slot. In such cases, we combine multiple physical slots to implement a virtual slot.

5 Dynamic Memory Allocation

We classify reg-stateful applications on a programmable network into elastic and inelastic applications based on whether an application can work with a variable amount of register memory. An inelastic application requires a fixed amount of register memory; it cannot work with less (e.g., NetChain [22]). An elastic application does not have a fixed register memory requirement. Our key observation is that most elastic applications overcome insufficient register memory with a fallback mechanism to the network control plane or the servers [23, 47]. The amount of memory typically affects application-level performance metrics (e.g., the system throughout in NetCache [23]). Although it may be possible to transform inelastic applications to elastic ones [29], we leave that to application developers. NetVRM supports both types, while only elastic applications can benefit from NetVRM's dynamic memory allocation.

Each application is specified with four parameters: the *application type* (e.g., HH); the *subnet* in which the application will run (e.g., 10.0.0.0/8); the *utility metric*, which is either the default metric (i.e., memory hit ratio) or an application-specific one; and the *utility target* (e.g., 0.98 for memory hit ratio). For an inelastic application, the amount of required memory is specified instead of the utility metric and target. NetVRM allocates the memory to it if the requirement can be satisfied, and rejects the application otherwise.

Dynamic memory allocation is only performed for elastic applications. NetVRM periodically polls the counters from the data plane, obtains the utility of each application, and dynamically allocates the register memory between the applications based on their utilities. There is a long line of work related to network utility maximization [26, 38, 40]. NetVRM presents three particular challenges for network utility maximization, including how to define the application utility properly, how to approximate the utility functions, and how to allocate the register memory in the network, which will be demonstrated in detail as follows.

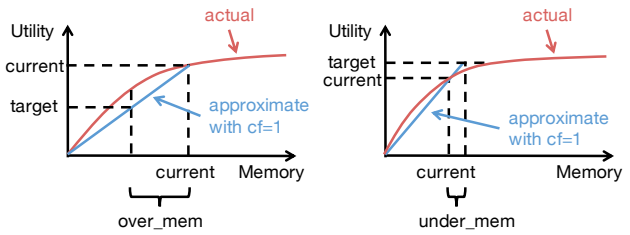


Figure 3: Utility function estimation.

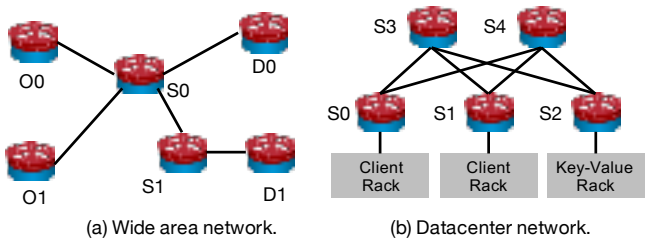


Figure 4: Examples for network-wide allocation.

5.1 Definition of Application Utility

Finding a proper definition of application utility is challenging, because different applications have their own application-level objectives that cannot be directly compared with each other (e.g., accuracy for a heavy-hitter detector or throughput for NetCache). NetVRM allows applications to compute their own utility metrics and report them to the allocator. Because not all the application-level metrics can be reported online (e.g., accuracy for a heavy-hitter detector), NetVRM also provides a default, generic utility definition. It is based on the observation that for many elastic applications, a register memory miss in packet processing usually affects the application-level performance, e.g., extra latency to process a packet with the fallback mechanism. Therefore, one effective utility definition is the memory hit ratio, which is the ratio of packets directly processed by the register memory in the switch. Besides being application-agnostic, this utility can be computed by tracking counters for memory hits in the data plane by NetVRM itself (§4). Moreover, the memory hit ratio is also a widely-used metric to evaluate the workload reduction for the fallback mechanism in many elastic applications on programmable networks [18, 39, 47].

5.2 Problem Formulation

We denote the available virtual register memory size of c switches in the network as $M_1, M_2, ..., M_c$, respectively. There are l applications running in the network. Let $i.target$ be the utility target of application i, and $i.utility(i.m_1, ..., i.m_c, i.T)$ be the utility function of application i where $i.m_j$ is the memory usage of application i on switch j and $i.T$ is the real-time traffic of application i. The network resource allocation problem is formulated as follows.

$$max \sum_{i=1}^{l} \mathbf{1}(i.utility(i.m_1, ..., i.m_c, i.T) \geq i.target)$$

$$\text{s.t.} \sum_{i=1}^{l} i.m_j \leq M_j, \forall j = 1, ..., c$$

The objective is to maximize the number of applications of which the utility targets can be satisfied, and the constraint is to ensure the sum of allocated memory on each switch does not exceed its memory size. We remark that this is one objective that is provided by default and has been used in several network management scenarios [36, 37]. NetVRM also supports other objectives and memory allocation algorithms.

Main challenge: unknown and dynamic utility functions. The main challenge to solve the allocation problem is that the utility functions of the applications are unknown and change over time. It is true that some utility functions can be known as a priori, e.g., the worst-case accuracy and the memory requirement for sketch-based heavy hitter detection (SHH) using count-min sketch [12] can be calculated mathematically [54]. But utility functions for many applications such as HH, NO and SS (§2) are hard to know in advance. More importantly, the solution needs to adapt to real-time traffic and as the applications are started and stopped dynamically.

Solution: online utility curve estimation without application knowledge. In order to adapt memory allocation for the applications *without* knowing their utility function, NetVRM leverages the observation that the utility function follows diminishing returns, i.e., that it is concave, which holds for a wide range of reg-stateful network applications [18, 23, 34, 47, 58], and approximates the memory requirement for each application. Let $i.util$, $i.target$ and $i.mem$ be the current utility, the utility target and the current memory for application i, respectively. The utility function is approximated by a polynomial function that intersects the origin. For an application i above its utility target, we use

$$i.over\_mem \leftarrow i.mem - (\frac{i.target}{i.util})^{cf} * i.mem \quad (1)$$

to estimate the amount of memory that can be moved from i to other applications ($i.over\_mem$). Because of *diminishing returns*, the utility function is *concave* and a linear function (when $cf = 1$) may underestimate $i.over\_mem$ (Figure 3(a)). We use a compensation factor cf which is set to be larger than 1 to compensate this. For an application i below its utility target, we use

$$i.under\_mem \leftarrow (\frac{i.target}{i.util})^{cf} * i.mem - i.mem \quad (2)$$

to estimate the amount of memory to be added to i ($i.under\_mem$). We use a cf larger than 1 for faster convergence (Figure 3(b)).

5.3 Network-Wide Register Memory Allocation

Based on the approximation in §5.2, NetVRM uses an online algorithm to move memory from over-provisioned applications (those above their utility targets) to under-provisioned applications (those below their utility targets) to maximize

Algorithm 1 Network-wide memory allocation

```
1:  new_plan ← cur_plan.copy()
2:  for application i in applications do
3:      if i.util ≥ i.target then
4:          satisfied_list.append(i)
5:          i.over_mem ← i.mem − (i.target/i.util)^cf ∗ i.mem
6:          distributed i.over_mem to i.paths proportionally
7:      else
8:          unsatisfied_list.append(i)
9:          i.under_mem ← (i.target/i.util)^cf ∗ i.mem − i.mem
10:         distributed i.under_mem to i.paths inverse proportionally
11: sort satisfied_list by over_mem in decreasing order
12: sort unsatisfied_list by i.under_mem in increasing order
13: for application i in unsatisfied_list do
14:     for path p in i.paths do
15:         sort p.switches based on i's existence and s.over_mem
16:         for switch s in p.switches do
17:             allocate memory from satisfied_list to p.under_mem
18:     if all paths are satisfied then
19:         update new_plan
20:     else
21:         move memory back to satisfied_list
22: return new_plan
```

the objective. The allocation are performed periodically to handle real-time traffic dynamics and application changes.

Main challenge: multiple and overlapped paths of an application. Besides the unknown and dynamic utility functions, the network-wide allocation problem is further complicated by the following two challenges. First, an application may need to handle traffic between multiple origin-destination (OD) pairs, and the traffic between each OD pair may use multiple paths. For example, in a wide area network, the operator may want to detect heavy hitters for flows between multiple OD pairs, e.g., O0-D0 and O1-D1 in Figure 4(a). In a datacenter network, the operator may want to provide in-network caching for traffic from multiple client racks to a key-value store rack, e.g., S0-S2 and S1-S2 in Figure 4(b). Datacenter networks typically use multi-path routing, e.g., path S0-S3-S2 and path S0-S4-S2 for traffic between S0 and S2. Second, different paths of an application may overlap, and thus can share their allocated memory. For example, in Figure 4(b), NetCache can be placed in S2 to save memory instead of in both S3 and S4.

Solution: network-wide memory allocation. At a high level, NetVRM performs network-wide memory allocation in two steps. First, NetVRM uses the utility estimation mechanism in §5.2 to estimate the required memory for each application, and decomposes *over_mem* or *under_mem* of each application to multiple paths. Second, it moves the memory from over-provisioned applications to under-provisioned applications. The pseudocode is shown in Algorithm 1.

The first step is to compute and decompose *over_mem* or *under_mem* of each application to multiple paths (line 2-10). NetVRM measures the utility (i.e., the memory hit ratio by default) and the traffic on each path. With the memory hit ratio as the utility, the utility (memory) of application *i* is the weighted average of its utilities (memories) by the traffic volume on its paths. We use the utility estimation mechanism in §5.2 to estimate *i.over_mem* and *i.under_mem*. Then *i.over_mem* is distributed to each path in proportional to their traffic (line 6) and *i.under_mem* is distributed to each path in inverse proportional to their traffic (line 10). We remark that NetVRM also allows disproportional memory allocation.

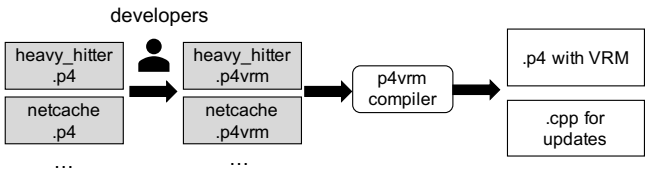


Figure 5: P4VRM compiler compiles P4VRM programs.

The second step is to move memory from over-provisioned applications to under-provisioned applications (line 11-21). We use a heuristic that reduces the memory for applications that are more over-provisioned first, and allocates the memory to the applications that are more likely to be satisfied first (line 11-12). For each unsatisfied application, it tries to satisfy the estimated memory requirement on each path (line 13-21). Because each path contains several switches, the algorithm needs to decide which switch to allocate memory from to satisfy the application (line 15-17). Two factors are considered in the decision, which are whether the application already has memory allocated on a switch (i.e., *i*'s existence) and how much extra memory a switch has (i.e., *s.over_mem*). These factors aim to avoid small amounts of memory scattering in many switches. If the application's requirement can be satisfied, the plan is updated (line 18-19). Otherwise, the memory is moved back to the satisfied applications (line 20-21).

To accommodate path overlapping, two extensions are required to the algorithm. First, in the utility estimation, the memory on overlapping switches is counted once for each overlapping path. Second, in memory allocation, the memory allocated to an application on overlapping switches is also counted once for each overlapping path.

Admission control, drop and priority. Admission control is critical when the total memory requirement exceeds the register memory size in the network. NetVRM admits one application into the network only if there is more available memory on each path than a predefined fraction of the total memory. NetVRM drops one application if it cannot meet the utility target in multiple consecutive allocation epochs. NetVRM targets elastic applications which can work even with no register memory. Thus, *if one application is rejected or dropped, it can turn to the fallback mechanism.* A malicious application with a tough utility target to satisfy would likely be dropped after a few allocation epochs. The operator can also assign custom priorities for the applications. For example, an application can be configured to not be dropped, or be assigned with a minimal amount of memory to avoid starvation when it is under-provisioned.

⟨*p4_declaration*⟩ ::= ⟨*vrm_reg_declaration*⟩ | ⟨*vrm_blb_declaration*⟩ | ...

⟨*vrm_reg_declaration*⟩ ::= 'vrmReg' ⟨*virt_stage*⟩ ⟨*register_declaration*⟩

⟨*vrm_blb_declaration*⟩ ::= 'vrmMergeable' ⟨*blackbox_declaration*⟩
| 'vrmNonMergeable' ⟨*blackbox_declaration*⟩

⟨*table_declaration*⟩ ::= ...
| 'vrmMergeable' ⟨*virt_stage*⟩ ⟨*table_declaration*⟩
| 'vrmNonMergeable' ⟨*table_declaration*⟩

⟨*action_function_declaration*⟩ ::= ...
| 'vrmMergeable' ⟨*action_function_declaration*⟩
| 'vrmNonMergeable' ⟨*action_function_declaration*⟩

⟨*control_statement*⟩ ::= ...
| 'HIT_COUNTER;'
| 'PKT_COUNTER;'

⟨*virt_stage*⟩ ::= ⟨*decimal_value*⟩

Figure 6: The P4VRM extensions to the P4-14. Gray non-terminal nodes refer to legacy rules in P4-14.

Memory reallocation process. At the end of each allocation epoch, NetVRM fetches the counters from the control plane, and computes the online utilities and the new memory allocation plan. Updating the memory allocation plan results in remapping from virtual addresses to physical addresses and moving existing entries because of the remapping. There are general solutions that can be applied to ensure the consistency of memory allocation updates [24, 53]. We apply two optimizations for particular cases in NetVRM. First, network measurement applications periodically reset the state such as counters maintained by the register memory. We align the memory allocation updates with the resetting operations, so that the memory allocation can be updated without moving existing entries and does not scarifice application correctness. Second, network applications that use lookup-table-based address translation can simply use a delta update when the memory size decreases, and allow more entries when the memory size increases. This ensures consistency because a lookup table is used for maintaining each address mapping.

6 Language Extension and Autogeneration

NetVRM provides P4VRM, an extension to the basic syntax and semantics of the P4 programming language [8] that supports virtual register memory abstraction and online utility measurement. Our implementation is based on P4-14, as more existing implementations are implemented in this version, but the same extensions could be applied to P4-16 as well. As shown in Figure 5, to port existing *.p4* programs, developers extend them to *.p4vrm* programs by marking which register arrays are to be virtualized and adding the online utility measurement primitives (`HIT_COUNTER` and `PKT_COUNTER`) correctly according to the applications. The P4VRM compiler takes multiple *.p4vrm* programs as input and outputs one merged P4 program (for the data plane) with virtual memory abstraction and online utility measurement, together with the C++ APIs (for the control plane) to configure the virtual register memory efficiently.

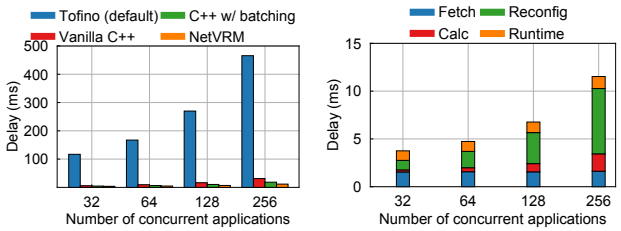
```
+ #include "params.p4"
- vrmReg 1 register stg1 {
+ register virtual_stg1 {
    width:32;
- instance_count:8192;
+  instance_count:65536;
   }
- vrmNonMergeable blackbox stateful_alu salu_stg1 {
+ blackbox stateful_alu salu_stg1 {
-  reg: stg1;
+  reg: virtual_stg1;
   ...
  }
- vrmNonMergeable action act_stg1() {
+ action act_stg1() {
- salu_stg1.execute_stateful_alu_from_hash(hash_1);
+ salu_stg1.execute_stateful_alu(params_md.slot_idx);
  }
- vrmNonMergeable table tbl_stg1 {
+ table tbl_stg1 {
+    actions {act_stg1;};
+    default_action:act_stg1();
+ }

  control ingress {
    if (valid(tcp) or valid(udp)) {
+       apply(set_app_id);
+       apply(set_offset_hf);
+       apply(add_offset);
+       if (params_md.app_type==0) {
           apply(tbl_stg1);
           ...
-          HIT_COUNTER;
+          apply(hit_counter);
           ...
-          PKT_COUNTER;
+          apply(pkt_counter);
+       }
    }
  }
```

Figure 7: An example of P4VRM code transformation by P4VRM compiler. '-' and '+' annotate the change before and after the transformation, respectively.

Grammar. As shown in Figure 6, P4VRM extends the P4-14 language specification [2] by introducing new keywords (`vrmReg`, `vrmMergeable` and `vrmNonMergeable`) to tag declarations related to a register array (`register`, `blackbox`, `action`, and `table`). It marks the register array as virtualized, and marks the related blackboxes, actions and tables that have the same logic as mergeable. It also specifies the stages at which the mergeable tables should be placed (`virt_stage`). The two primitive statements (i.e., `HIT_COUNTER` and `PKT_COUNTER`) are used for online utility measurement. `HIT_COUNTER` tracks the number of packets processed by the register memory, and `PKT_COUNTER` tracks the total number of packets of the application.

Generating merged P4 programs and C++ APIs. To merge parsers, P4VRM compiler abstracts the packet parser of each application as a Finite State Machine (FSM) and merges the identical states into a single FSM. Then, the P4VRM compiler transforms P4VRM-introduced declarations (i.e., `vrmReg`, `vrmMergeable` and `vrmNonMergeable`) to P4-14 declarations (i.e., `register`, `blackbox`, `action` and `table`), and adds the additional logic for address trans-



(a) Total control loop delay vs. different implementations. (b) Delay breakdown for NetVRM.

Figure 8: Analysis of control loop delay.

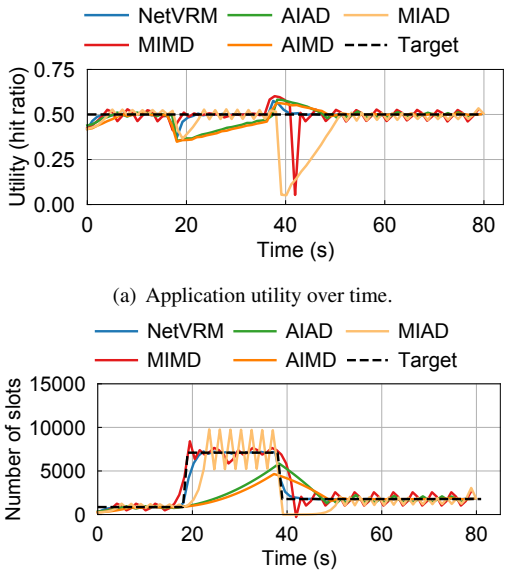
lation, as shown in Figure 7. The compiler also loads a P4 library (`params.p4`) provided by P4VRM, containing additional metadata and logic (e.g., to perform the page table lookup) and adds the appropriate invocations at the beginning of the pipeline. Finally, the compiler generates control plane APIs for resetting counters, fetching counters, resetting virtual memory and configuring the virtual memory.

Requirement for merge. Merging multiple reg-stateful applications needs to comply with the same resource constraints as in existing work [19, 56, 57], most notably those related to register memory (e.g., total register memory size per stage, stateful ALUs per stage). If merging violates hardware constraints, the P4VRM compiler would fail and produce no output.

7 Implementation

We have implemented a NetVRM prototype on a 6.5 Tbps Intel Tofino switch [50], and used commodity servers to replay traces and generate traffic. The P4 library we provide for virtual register memory support is around 500 lines of P4-14 code. The virtual register memory spans eight physical stages. Other stages are used for necessary functionalities (e.g., routing and enabling concurrent applications). We emulate four switches with the four independent pipelines of the Tofino switch. The data plane program decides which emulated switch one packet enters by checking the ingress port. The implementation batches the data plane updates, and uses multithreading to update the four pipelines simultaneously. The NetVRM control plane implementation consists of around 2200 lines of C++ code. The P4VRM compiler is built on Flex/Bison [31] and parses the .p4vrm files to build an AST. It consists of around 2000 lines of C++ and 900 lines of grammar.

Overhead of NetVRM. The address translation needs to be done in two stages (§4), which is realized with two tables to adjust the `slot_idx` (Figure 7). The first table (`set_offset_hf`) can be placed in the same stage with other tables (e.g., `set_app_id`) that are necessary and inevitable for concurrent applications running. The register memory in the second stage where `add_offset` is placed cannot be virtualized, which can be used for basic networking functionality and inelastic applications. In some cases, the register memory in some stages cannot be used even without



(a) Application utility over time.

(b) Register memory consumption over time.

Figure 9: Comparison of different algorithms to update memory allocation.

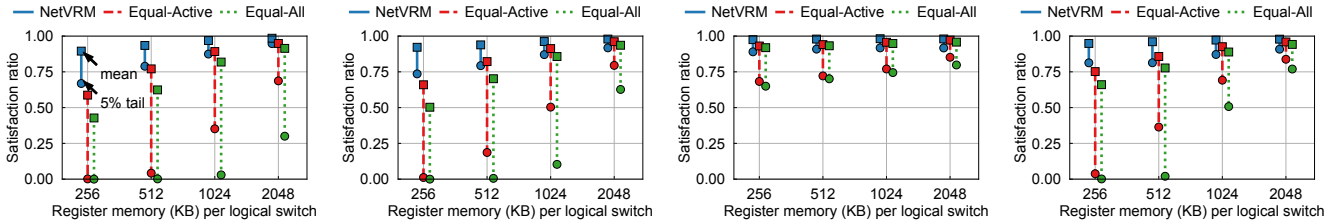
NetVRM because of the indivisibility between the virtual slot size and the number of stages. For example, if there are three physical stages available for virtualization, an application with 2-stage virtual slots can use two stages at most. Then the page table placed in the first stage does not introduce extra register memory overhead. We remark that this is a common case for many applications [18, 23, 34, 47, 54]. The extra resource needed by each application in NetVRM is only one table entry in the page table and two counters for the online utility measurement, without extra stage overhead.

8 Evaluation

We evaluate our NetVRM prototype in two scales. We first use microbenchmarks to examine the control loop delay and the properties of the resource allocation algorithm (i.e., stability and convergence speed). With macrobenchmarks, we demonstrate the benefits of NetVRM in combination with a variety of network applications, workload parameters, comparisons with alternative approaches and network topologies.

8.1 Microbenchmark

Control loop delay. We emulate four switches by the four independent pipelines of the Tofino switch. First, we compare the total control loop delay, i.e., the time to complete a virtual memory reallocation (§5.3), with different implementations, including the default implementation on Tofino switches which uses Python Thrift APIs, a vanilla C++ implementation, a C++ implementation with batching, and NetVRM, which incorporates both batching and multithreading. As shown in Figure 8(a), the C++ implementations are an order of magnitude more efficient than the default implementation of Tofino control plane APIs. NetVRM's optimizations further reduce the delay by a factor of ~ 3.



(a) Heavy hitter detection (HH). (b) Newly opened TCP (NO). (c) Superspreader detection (SS). (d) Mix of HH, NO, SS.

Figure 10: Satisfaction for flow-based applications in the WAN scenario.

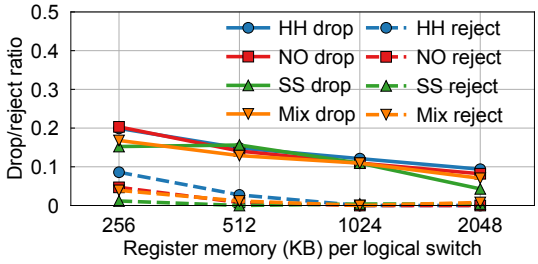


Figure 11: Drop/reject ratio of NetVRM for flow-based applications in the WAN scenario.

We further break down the control loop delay of NetVRM into four parts, i.e., *Fetch*, *Calc*, *Reconfig* and *Runtime*, and measure the latencies with different number of concurrent applications. *Fetch*, *Calc*, *Reconfig* and *Runtime* represent the time of fetching counters, calculating online utility and new memory allocation plan, configuring the page table, and the runtime overhead for resetting the state (e.g., the counters), respectively. As shown in Figure 8(b), the time of *Fetch* remains relatively constant since we use batching to fetch all the counters together where the data size does not influence the latency significantly. The time of *Calc* increases with more applications, due to the heavier overhead to compute the online utility and memory allocation plans. The time of *Reconfig* dominates the control loop delay because of the intensive updates to the data plane for four pipelines.

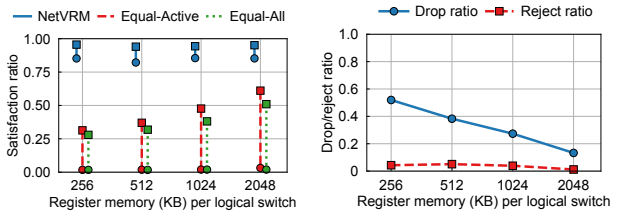
Due to the limit of our testbed, we only emulate four switches with one Tofino switch in our experiment. We remark that NetVRM can maintain the low control loop delay and scale in real wide area networks and datacenters with a larger number of switches for two reasons. First, *Fetch*, *Reconfig* and *Runtime*, which do not need coordination between multiple switches, can be done in different switches locally and simultaneously. Second, *Calc* needs to compute the online network-wide utility and memory allocation plans for each application which has to be done in a centralized location. Instead of doing it on the switch OS with limited computation capability in our experiment, the time of *Calc* can be reduced easily by running it in a more powerful server.

Stability and fast convergence of NetVRM. In this experiment, we compare NetVRM with other alternative approaches which are commonly used in network resource allocation, including AIAD, MIAD, MIMD and AIMD. Those approaches estimate the memory requirements by increasing (decreasing) the step size additively (A) or multiplicatively (M) when the satisfaction status remains the same (changes) compared with the previous epoch. We run one NetCache [23] application on the switch and set its memory hit ratio target to be 0.5. The workload skewness is Zipf-0.99 at the beginning, then changes to Zipf-0.9 at 18 seconds, and finally changes to Zipf-0.95 at 38 seconds. Figure 9(a) and Figure 9(b) show the utility and memory usage over time, respectively. AIAD and AIMD fail to meet the utility target when the skewness becomes Zipf-0.9 because increasing the memory additively is too slow. MIAD converges slower after 38 seconds because decreasing the step size additively from a large step size is slow. MIMD has the closest performance to NetVRM, but the utility fluctuates around the utility target after convergence. NetVRM estimates the memory requirements based on the online utility (§5.2). Thus, it can react fast and more accurately to the traffic dynamics and maintain the utility above its target most of the time.

8.2 Macrobenchmark

NetVRM configuration and network topology. The default allocation epoch and measurement epoch are both one second. The default network topology is the Wide Area Network (WAN), where each application has traffic from 4 switches independently. NetVRM drops an application if it cannot meet the utility target in four consecutive epochs and rejects an application if the available memory on the switch is smaller than 1/128 of the total memory.

Network applications. NetVRM supports a wide range of network applications. We use five applications in the evaluation, i.e., heavy hitter detection (HH) [47], newly opened TCP connection detection (NO) [55], superspreader detection (SS) [47], sketch-based heavy hitter detection (SHH) [54] and NetCache [23]. HH, NO and SS are flow-based applications which store precise flow records on the data plane, and evict the existing entries to the control plane upon hash collisions, following the eviction policy in TurboFlow [47]. SHH is a sketch-based application that uses approximate data structures (i.e., count-min sketch [12]) to approximate flow records. NetCache maintains hot key-value pairs on the data plane to serve a request upon a cache hit. For each application type, there can be multiple instances of this application, e.g., belonging to different clients/tenants. Each client/tenant owns



(a) Satisfaction. (b) Drop/reject ratio.

Figure 12: Experimental results for sketch-based applications (SHH) in the WAN scenario.

a /8 subnet of source IP, and can dynamically start or stop application instances within its subnet.

Traffic traces. The traces for measurement applications on WAN are the 2019 passive CAIDA traces [10]. The datacenter traces are from Facebook's production clusters [41]. We replay the traces via MoonGen [15]. The NetCache traffic is generated by our DPDK client according to the Zipf distribution with different skewness parameters.

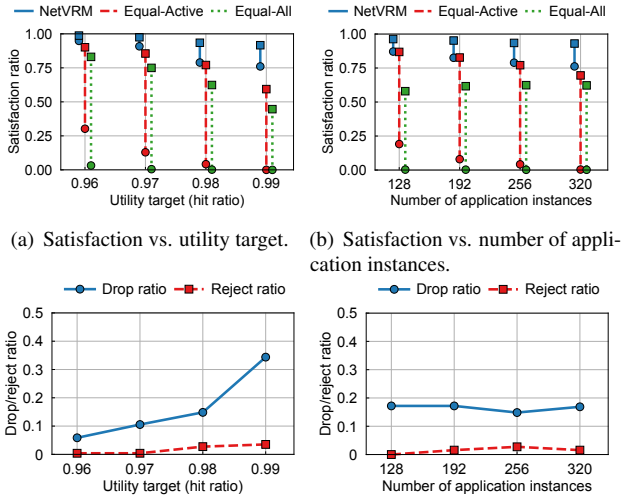
Alternative approaches. We compare NetVRM with two alternative approaches. (i) One is *Equal-All*, which *statically* assigns an equal amount of register memory to all applications, active or not. For example, if each application instance runs within a /8 subnet, then there are at most 256 concurrent application instances. Thus, *Equal-All* assigns 1/256 of total memory to each instance. (ii) The other is *Equal-Active*, which only assigns an equal amount of register memory to *active* instances. We emphasize that *Equal-Active* is enabled by the ability of NetVRM to dynamically allocate register memory at runtime. NetVRM further improves *Equal-Active* with the network-wide memory allocation algorithm in §5.

Performance metric. We use *satisfaction ratio* as the performance metric for these network applications. Each application instance has a utility target. The satisfaction ratio of an instance is the fraction of time the utility target is met during its lifetime. For each experiment, we compute the satisfaction ratio for every instance, and show the mean and 5th percentile of the satisfaction ratios across all instances. Considering the number of instances is only a few hundreds (i.e., 256), the 5th percentile catches the tail pattern in the last ten instances, while other options (e.g., 1th, 0.1th) are too limited which only show the satisfaction of the last one or two instances.

8.2.1 Generality

We show that NetVRM is general to a wide range of network application types in the WAN scenario.

Setup. We replay the CAIDA traffic on the four emulated switches as in §8.1. We deploy four types of applications including HH, NO, SS and SHH. We omit NetCache as it is not a good use case for the WAN scenario. HH maintains the flow records of the source IP and the corresponding number of packets for all the IP traffic. NO maintains the flow records of the source IP and the corresponding number of packets only for TCP SYN packets. SS records the distinct IP address



(a) Satisfaction vs. utility target. (b) Satisfaction vs. number of application instances.

(c) Drop/reject ratio vs. utility target. (d) Drop/reject ratio vs. number of application instances.

Figure 13: Impact of workload parameters.

pair (source IP and destination IP) for all the IP traffic. SHH maintains the flow records of the source IP and the threshold to be identified as a heavy hitter is set to 200. We do the following extension for a network-wide SHH: one SHH's utility is defined as the smallest worst-case accuracy across its switches. Since each stage only supports 32-bit read and write from register memory on the data plane, each virtual slot of the three applications spans two physical stages and there are up to 256K virtual slots (i.e., 2048 KB register memory) on each switch.

By default, there are 256 application instances started in 20 minutes based on a Poisson Process and the running time of the instances follows a uniform distribution from 6 minutes to 14 minutes. The utility targets are specified by the operator based on operational requirements. The default utility target for HH, NO, SS, i.e., the memory hit ratio, is 0.98, and the default utility target for SHH, i.e., the worst-case accuracy, is 0.98. On each switch, we use a /8 instance filter and a /2 switch filter to identify the traffic to be processed by each instance. We feed the CAIDA traces into four switches simultaneously and measure the mean and 5th percentile of satisfaction across the 256 instances.

We remark that *this is only one setup of a demanding workload to stress the system*, following the similar workload pattern in [36, 37]. We show that NetVRM outperforms the alternatives with different workload parameters in §8.2.2.

Results. Figure 10 shows the satisfaction ratios for flow-based applications (i.e., HH, NO, SS) under different amounts of register memory. For each vertical line, the upper square end is the mean satisfaction ratio, and the lower round end is the 5th percentile satisfaction ratio, among the 256 application instances. Figure 10(a), (b) and (c) show the cases that the instances are from the same application type, and (d) shows the case that the instances are from all the three types.

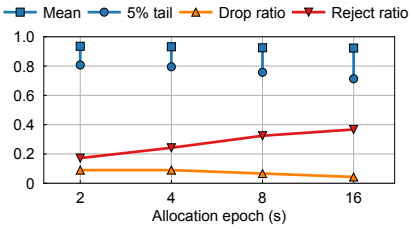
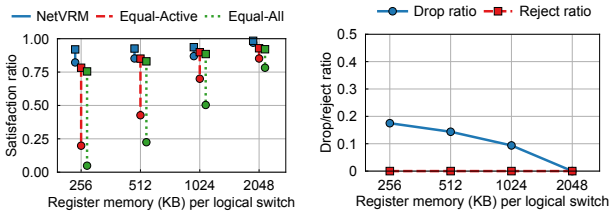


Figure 14: Impact of allocation epochs on NetVRM.



(a) Satisfaction. (b) Drop/reject ratio.

Figure 15: Experimental results in the datacenter scenario.

When the register memory is limited (e.g., 256 KB), NetVRM significantly outperforms *Equal-All* and *Equal-Active* on both the mean and the tail. When the register memory is abundant (e.g., 2048 KB), NetVRM is able to maintain both high mean and 5th percentile satisfaction ratios. In contrast, *Equal-All* and *Equal-Active* have comparable mean satisfaction ratios, but suffer from the tail behavior. The advantage of NetVRM over *Equal-All* and *Equal-Active* is consistent across different application types. SS uses src IP and dst IP as the hash key. Thus, it has fewer hash collisions than HH and NO, leading to a higher satisfaction ratio. Figure 11 shows the drop ratios and rejection ratios of NetVRM under the four workloads. Similarly, SS drops and rejects fewer application instances than HH and NO, because it has fewer hash collisions and less memory requirement.

Figure 12 shows that NetVRM outperforms *Equal-All* and *Equal-Active* with the sketch-based applications (i.e., SHH) as well. Compared with flow-based applications, the alternatives have lower satisfaction ratios and NetVRM drops more application instances because SHH needs more memory to guarantee the worst-case accuracy bounds.

The alternatives, *Equal-All* and *Equal-Active*, have close performance for all the applications, which means only having the mechanism of virtual register memory to allocate resources to active applications is not sufficient. The allocation algorithm that decides the memory allocation plan is critical to the performance.

8.2.2 Analysis of NetVRM

We analyze NetVRM by showing the impact of workload parameters and the allocation epoch. We use the same setup in §8.2.1 and show the results for the workload of HH. The findings for other application types are similar. We demonstrate the benefits of NetVRM over the local memory allocation approach in Appendix B.

Impact of workload parameters. Figure 13(a) shows that NetVRM is able to manage the register memory efficiently with different utility targets. With more strict targets, the three approaches have worse performance as the application instances have higher memory requirements. Figure 13(c) shows the drop ratio and reject ratio increase with more strict targets. Figure 13(b) studies the impact of the number of application instances arriving in each experiment. Fewer instances mean less resource contention, leading to higher satisfaction. NetVRM consistently outperforms the alternatives. Interestingly, Figure 13(d) shows that the drop ratio and reject ratio are not significantly influenced by the number of instances in the evaluated range.

Impact of the allocation epoch. Figure 14 shows that a shorter allocation epoch leads to a slightly better performance, both in mean and tail. A longer allocation epoch can get a comparable satisfaction ratio but it comes with rejecting more applications. For example, when the allocation epoch is 16 seconds, NetVRM drops and rejects about 40% application instances, while the sum of drop ratio and reject ratio is 25% when the allocation epoch is 2 seconds.

8.2.3 NetVRM in Datacenter Network

Setup. We use the four independent pipelines of the Tofino switch to emulate four switches, and wire the four switches to build a datacenter network topology (shown in Appendix C). S0, S1 and S2 are ToR switches for client rack 1, client rack 2 and the key-value rack respectively. S3 is a spine switch connecting to them. We run two types of applications, which are HH and NetCache. HH records the number of packets of distinct four tuples (source IP, destination IP, source port, destination port). We use the Cluster-C traffic trace from Facebook's production datacenters [41]. The trace is anonymized by hashing. The IP addresses are hashed to 64 bits and the port numbers are hashed to 32 bits in the trace. The HH application uses six physical stages to store the four tuples and one extra stage to store the number of packets. We generate pcap files from the Facebook trace, and assign the timestamps of the packets uniformly in one second as the original timestamp is at second granularity. Each application instance owns a /8 subnet. There are 318 HH instances arriving in 20 minutes based on a Poisson process, and the running time of the instances follows a uniform distribution from 6 minutes to 14 minutes. The HH instances use two paths, S0-S3-S2 and S1-S3-S2. The utility target of HH is set to 0.96.

We run two NetCache instances. NetCache1 (NC1) uses path S0-S3-S2, and NetCache2 (NC2) uses path S1-S3-S2. The tenants of NC1 and NC2 are in client rack 1 and client rack 2, respectively, which access different key-value items in the key-value rack, so they cannot share the memory on S2 and S3. NC1 and NC2 run throughout the 30-minute experiment time. The workload skewness changes between Zipf-0.99 and Zipf-0.95 every 6 minutes. The utility target is 0.5. Each virtual slot of NetCache spans 8 physical stages,

resulting in up to 64K virtual slots per switch. The NetCache instances are set to not be dropped.

Results. Figure 15(a) shows the satisfaction ratios of the three approaches, and Figure 15(b) shows the drop ratios and reject ratios of NetVRM. Similarly, NetVRM outperforms *Equal-All* and *Equal-Active* consistently under different amounts of register memory. It indicates that NetVRM can multiplex the register memory between different switches in a complicated scenario where applications have multiple paths and measurement applications run along with datacenter-specific applications such as NetCache.

9 Conclusion

We present NetVRM, a network management system to support dynamic register memory sharing between multiple concurrent applications on a programmable network. NetVRM provides a *virtual register memory abstraction* that enables register memory sharing in the switch data plane, and dynamically allocates memory for better resource efficiency and application utility. NetVRM also provides P4VRM as an extension of P4 for developing applications with virtual register memory, and a compiler to generate data plane programs and control plane APIs.

Acknowledgments. We thank our shepherd Laurent Vanbever and the anonymous reviewers for their valuable feedback on this paper. Xin Jin (xinjinpku@pku.edu.cn) is the corresponding author. Xin Jin is with the Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education. This work is supported in part by NSF grants CNS-1813487, CCF-1918757 and CNS-2008048, and the National Natural Science Foundation of China under the grant number 62172008.

References

[1] In-band Network Telemetry (INT) Dataplane Specification. https://github.com/p4lang/p4-applications/blob/master/docs/INT.pdf.

[2] P4-14 Language Specification. https://p4.org/p4-spec/p4-14/v1.0.5/tex/p4.pdf.

[3] P4 Behavioral Model Repository. https://github.com/p4lang/behavioral-model.

[4] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS*, June 2012.

[5] Networking and Content Delivery on AWS. https://aws.amazon.com/products/networking/.

[6] Multitenant SaaS on Azure. https://docs.microsoft.com/en-us/azure/architecture/example-scenario/multi-saas/multitenant-saas.

[7] Azure networking services overview. https://docs.microsoft.com/en-us/azure/networking/fundamentals/networking-overview.

[8] 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 CCR*, July 2014.

[9] 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 *ACM SIGCOMM*, August 2013.

[10] The CAIDA Anonymized Internet Traces 2019 Dataset. https://data.caida.org/datasets/passive-2019/.

[11] Cavium XPliant. https://www.cavium.com/.

[12] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 2005.

[13] H. T. Dang, M. Canini, F. Pedone, and R. Soulé. Paxos made switch-y. *SIGCOMM CCR*, April 2016.

[14] H. T. Dang, D. Sciascia, M. Canini, F. Pedone, and R. Soulé. NetPaxos: Consensus at network speed. In *ACM SOSR*, June 2015.

[15] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle. Moongen: A scriptable high-speed packet generator. In *ACM SIGCOMM Conference on Internet Measurement Conference*, 2015.

[16] S. K. Fayaz, Y. Tobioka, V. Sekar, and M. Bailey. Bohatei: Flexible and elastic ddos defense. In {*USENIX*} *Security*, 2015.

[17] A. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi. Participatory networking: An API for application control of SDNs. In *ACM SIGCOMM*, August 2013.

[18] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger. Sonata: Query-driven streaming network telemetry. In *ACM SIGCOMM*, 2018.

[19] D. Hancock and J. Van der Merwe. Hyper4: Using p4 to virtualize the programmable data plane. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*, 2016.

[20] Q. Huang, P. P. Lee, and Y. Bao. Sketchlearn: Relieving user burdens in approximate measurement with automated statistical inference. In *ACM SIGCOMM*, 2018.

[21] X. Jin, J. Gossels, J. Rexford, and D. Walker. CoVisor: A compositional hypervisor for software-defined networks. In *USENIX NSDI*, May 2015.

[22] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica. NetChain: Scale-free sub-RTT coordination. In *USENIX NSDI*, April 2018.

[23] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica. NetCache: Balancing key-value stores with fast in-network caching. In *ACM SOSP*, October 2017.

[24] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer. Dynamic scheduling of network updates. In *ACM SIGCOMM*, August 2014.

[25] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford. Hula: Scalable load balancing using programmable data planes. In *ACM SOSR*, March 2016.

[26] F. Kelly and T. Voice. Stability of end-to-end algorithms for joint routing and rate control. *SIGCOMM CCR*, 2005.

[27] D. Kim, Z. Liu, Y. Zhu, C. Kim, J. Lee, V. Sekar, and S. Seshan. Tea: Enabling state-intensive network functions on programmable switches. In *ACM SIGCOMM*, 2020.

[28] D. Kim, J. Nelson, D. R. Ports, V. Sekar, and S. Seshan. Redplane: enabling fault-tolerant stateful in-switch applications. In *ACM SIGCOMM*, 2021.

[29] D. Kim, Y. Zhu, C. Kim, J. Lee, and S. Seshan. Generic external memory for switch data planes. In *ACM HotNets Workshop*, 2018.

[30] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, N. Gude, P. Ingram, et al. Network virtualization in multi-tenant datacenters. In *USENIX NSDI*, April 2014.

[31] J. Levine. *Flex & Bison: Text Processing Tools*. " O'Reilly Media, Inc.", 2009.

[32] M. Liu, L. Luo, J. Nelson, L. Ceze, A. Krishnamurthy, and K. Atreya. IncBricks: Toward in-network computation with an in-network cache. In *ACM ASPLOS*, April 2017.

[33] Z. Liu, Z. Bai, Z. Liu, X. Li, C. Kim, V. Braverman, X. Jin, and I. Stoica. Distcache: Provable load balancing for large-scale storage systems with distributed caching. In *USENIX FAST*, 2019.

[34] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *ACM SIGCOMM*, 2016.

[35] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *ACM SIGCOMM*, 2017.

[36] M. Moshref, M. Yu, R. Govindan, and A. Vahdat. DREAM: Dynamic resource allocation for software-defined measurement. In *ACM SIGCOMM*, August 2014.

[37] M. Moshref, M. Yu, R. Govindan, and A. Vahdat. Scream: Sketch resource allocation for software-defined measurement. In *ACM CoNEXT*, 2015.

[38] K. Nagaraj, D. Bharadia, H. Mao, S. Chinchali, M. Alizadeh, and S. Katti. Numfabric: Fast and flexible bandwidth allocation in datacenters. In *ACM SIGCOMM*, 2016.

[39] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim. Language-directed hardware design for network performance monitoring. In *ACM SIGCOMM*, August 2017.

[40] V. Nathan, V. Sivaraman, R. Addanki, M. Khani, P. Goyal, and M. Alizadeh. End-to-end transport for video qoe fairness. In *ACM SIGCOMM*, 2019.

[41] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the social network's (datacenter) network. In *ACM SIGCOMM*, 2015.

[42] A. Sapio, I. Abdelaziz, M. Canini, and P. Kalnis. Daiet: a system for data aggregation inside the network. In *ACM Symposium on Cloud Computing*, 2017.

[43] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. R. K. Ports, and P. Richtárik. Scaling distributed machine learning with in-network aggregation, 2019.

[44] M. Saquetti, G. Bueno, W. Cordeiro, and J. R. Azambuja. P4vbox: Enabling p4-based switch virtualization. *IEEE Communications Letters*, 2019.

[45] N. K. Sharma, A. Kaufmann, T. E. Anderson, A. Krishnamurthy, J. Nelson, and S. Peter. Evaluating the power of flexible packet processing for network resource allocation. In *USENIX NSDI*, March 2017.

[46] R. Sherwood, G. Gibb, K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar. Can the production network be the testbed? In *USENIX OSDI*, October 2010.

[47] J. Sonchack, A. J. Aviv, E. Keller, and J. M. Smith. Turboflow: Information rich flow record generation on commodity switches. In *EuroSys*, 2018.

[48] H. Soni, T. Turletti, and W. Dabbous. P4Bricks: Enabling multiprocessing using linker-based network data plane architecture. 2018.

[49] R. Stoyanov and N. Zilberman. Mtpsa: Multi-tenant programmable switches. In *Proceedings of the 3rd P4 Workshop in Europe*, 2020.

[50] Intel Tofino. `https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html`.

[51] T. Wang, H. Zhu, F. Ruffy, X. Jin, A. Sivaraman, D. R. K. Ports, and A. Panda. Multitenancy for fast and programmable networks in the cloud. In *USENIX HotCloud Workshop*, 2020.

[52] D. Wu, A. Chen, T. E. Ng, G. Wang, and H. Wang. Accelerated service chaining on a single switch asic. In *ACM HotNets Workshop*, 2019.

[53] L. Yu, J. Sonchack, and V. Liu. Mantis: Reactive programmable switches. In *ACM SIGCOMM*, August 2020.

[54] M. Yu, L. Jose, and R. Miao. Software defined traffic measurement with opensketch. In *USENIX NSDI*, 2013.

[55] Y. Yuan, D. Lin, A. Mishra, S. Marwaha, R. Alur, and B. T. Loo. Quantitative network monitoring with netqre. In *ACM SIGCOMM*, 2017.

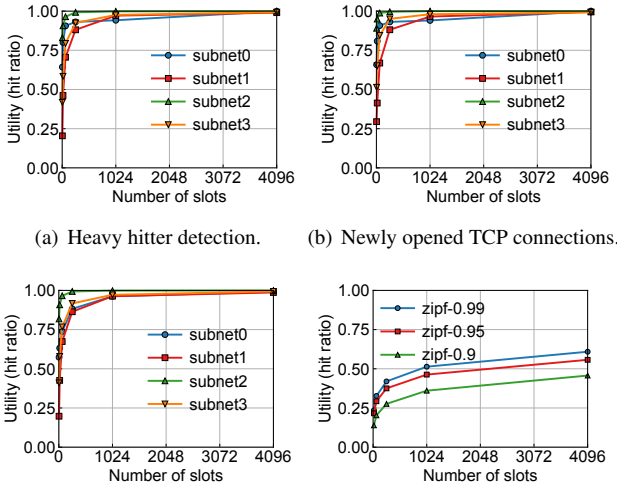
[56] C. Zhang, J. Bi, Y. Zhou, A. B. Dogar, and J. Wu. Hyperv: A high performance hypervisor for virtualization of the programmable data plane. In *2017 26th International Conference on Computer Communication and Networks (ICCCN)*, 2017.

[57] P. Zheng, T. Benson, and C. Hu. P4visor: Lightweight virtualization and composition primitives for building and testing modular programs. In *ACM CoNEXT*, 2018.

[58] H. Zhu, Z. Bai, J. Li, E. Michael, D. Ports, I. Stoica, and X. Jin. Harmonia: Near-linear scalability for replicated storage with in-network conflict detection. In *Proceedings of the VLDB Endowment*, November 2019.

A Diminishing Return Examples

Figure 16 demonstrates the diminishing returns for four applications. The first three are measurement applications: heavy hitter detection (HH) [54], newly opened TCP connection detection (NO) [55], superspreader detection (SS) [54]. These applications store flow records in the data plane; hash collisions caused by inadequate memory require additional control plane processing. The fourth, NetCache [23] caches hot objects in the switch data plane to improve the throughput of a key-value store. The utility is measured using memory hit ratio. We evaluate the measurement applications (Figure 16(a–c)) on traffic from different subnets of the 2019 passive CAIDA trace [10], and NetCache on a synthetic Zipf workload with different skewness parameters (Figure 16(d)).

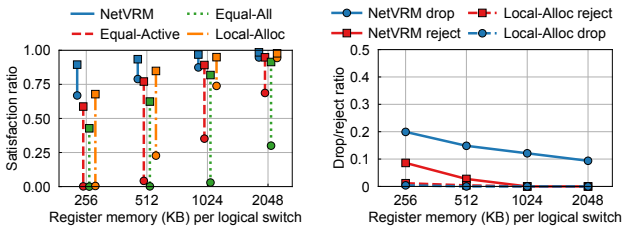


(a) Heavy hitter detection. (b) Newly opened TCP connections.

(c) Superspreader detection. (d) NetCache.

Figure 16: Examples for the diminishing returns of the utility curves in reg-stateful network applications.

B Additional Evaluation Results



(a) Satisfaction. (b) Drop/reject ratios of NetVRM and *Local-Alloc*.

Figure 17: Comparison with *Local-Alloc*.

Comparison with local memory allocation. Besides the *Equal-all* and *Equal-Active*, we also compare NetVRM with *Local-Alloc* which only does memory allocation and makes drop/reject decisions on individual switches locally. One application is counted as drop/reject only after all the four switches have decided to drop/reject it. We report the results for HH workload. The findings for other application types are similar. Figure 17 shows that *Local-Alloc* has better performance than *Equal-all* and *Equal-Active*, but is still worse than NetVRM because it fails to capture network-wide information and makes sub-optimal allocation and drop/reject decisions.

C Network Topology in Datacenter Scenario

We wire the four emulated switches to build a datacenter network topology, as shown in Figure 18, to evaluate the performance of NetVRM in the datacenter scenario.

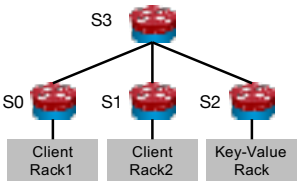


Figure 18: Datacenter topology for evaluation.