NutCracker: A Compilation Framework for Hybrid DPU Architectures

Yihan Yang[1], Haifeng Sun[1], Antoine Kaufmann[2], Jialin Li[1]
[1] National University of Singapore [2] MPI-SWS

Abstract

SoC-based SmartNICs, or data processing units (DPUs), are becoming a viable option for offloading infrastructure services. However, developers need deep hardware-level knowledge to fully utilize the available hardware accelerators on a target DPU. Hardware heterogeneity also makes porting across DPUs a formidable task. In this work, we propose a new compiler framework, NutCracker. Using NutCracker, programmers develop DPU applications using high-level target-independent languages. NutCracker applies a two-stage compilation process. It first performs progressive lowering to convert the source program to candidate intermediate representations (IRs) of the target DPU. Next, NutCracker applies cost-guided mapping optimization using equality saturation to select a final implementation on the target hardware with a configurable optimization goal. Evaluated on eight applications, NutCracker reduces developer effort by nearly 90% while delivering performance within 3% of handcrafted implementations for seven of the workloads. Moreover, its compilation time is comparable to standard toolchains such as GCC.

CCS Concepts: • **Networks** → **Programming interfaces**; **Programmable networks**.

Keywords: Data-plane programming language, In-network computation

ACM Reference Format:
Yihan Yang, Haifeng Sun, Antoine Kaufmann, Jialin Li. 2026. NutCracker: A Compilation Framework for Hybrid DPU Architectures. In *European Conference on Computer Systems (EUROSYS '26), April 27–30, 2026, Edinburgh, Scotland Uk*. ACM, New York, NY, USA, 21 pages. https://doi.org/10.1145/3767295.3803618

1 Introduction

Host CPUs are becoming a scarce resource, particularly in cloud data centers. Every CPU cycle saved from infrastructure services can be converted to profit by renting to cloud tenants. With the "free-lunch" from processor scaling laws long gone, cloud providers are turning to hardware offloading solutions to save CPU resources from these services. Among various hardware platforms, SmartNICs have become a popular target for infrastructure service offloading. By incorporating processing elements directly on the host network data path, SmartNICs have enabled production-scale offloading of network virtualization [13], virtual machine hypervisors [8, 59], network functions [23, 43], and distributed computations [20, 41] in large cloud providers.

Developing applications for SmartNICs, however, has been notoriously difficult. Programming FPGA-based SmartNICs requires deep hardware expertise, making them a viable option only for the few organizations with large, experienced hardware teams. The recent line of Systems on a Chip (SoC)-based SmartNICs, also known as data processing units (DPUs), aims to reduce this programming burden. DPUs include an array of power-efficient general-purpose cores to run offloaded applications. The premise is that DPUs can be programmed similarly to regular host software, which both reduces development barriers and improves portability across DPU hardware.

The reality of DPU development is far from these lofty promises. The SoCs on modern DPUs are complex pieces of hardware. In addition to general-purpose CPU cores, these SoCs contain a diverse set of special-purpose accelerators, such as network processing units, packet processing pipelines, engines for RDMA and NVMe, and accelerators for various computations. These ASIC accelerators operate at extreme computational efficiency but are limited in programmability. In fact, using the wimpy general-purpose cores alone often falls short of attaining acceptable offloading performance in many scenarios. Unleashing the full processing capability of DPUs requires proper utilization of *all* available hardware resources. Unfortunately, doing so requires developers to gather sufficient knowledge of the functionality, resource availability, and performance profile of each hardware component on the target DPU. The process usually involves a deep learning curve and careful profiling efforts. Even worse, commercial DPUs [3, 12, 32] show significant heterogeneity in their hardware components. Hardware-level expertise in one DPU typically does not transfer to other DPU platforms.

In this work, we propose NutCracker, a compiler framework that eases the burden of developing high-performance applications on DPUs. NutCracker takes programs written

EUROSYS '26, Edinburgh, Scotland Uk

ACM ISBN 979-8-4007-2212-7/26/04
https://doi.org/10.1145/3767295.3803618

in high-level languages, such as C and P4 [6], and generates efficient backend code to run on a heterogeneous set of DPUs. The input programs are *hardware independent*; developers are not required to possess a deep understanding of the hardware or tailor their programs to a specific DPU.

NutCracker applies a two-stage compilation process. In the first phase, NutCracker lowers the input program to intermediate representations (IR) of the target DPU platform. However, specifying the lowering rules for every possible DPU platform is an intractable problem. We address this challenge by proposing *progressive lowering*. Despite their apparent heterogeneity, we observe that accelerators on DPUs can generally be grouped into three categories: fixed-function accelerators, match-action pipelines, and datapath processors. Accelerators within each category share similar high-level architectures and differ only in resource constraints and component functional semantics. Based on this insight, we propose *virtual Domain-Specific Architectures* (vDSAs) which capture the common architectural features of each accelerator category, without exposing hardware details. We then define a general MLIR [22] dialect for each vDSA, as well as rewriting rules to lower LLVM [21] IR and P4MLIR [35] to the vDSA dialects. vDSA allows us to define one set of rewriting rules for all DPU platforms. To further facilitate the rule generation process, we apply Egglog [58] and DialEgg [56] to define these rewriting rules declaratively.

Next, NutCracker applies fine-grained rewriting to further lower the vDSA IRs to the MLIR dialects of the target DPU platform. These target-specific dialects are defined by hardware vendors who have detailed knowledge of the hardware details. Our vDSA-based approach significantly reduces the development effort of these dialects. Each vendor only needs to define hardware constraints on the respective vDSA for each concrete accelerator type, a much simpler task than developing the entire set of MLIR rewriting rules. These constraints are also written declaratively, using DialEgg, to further ease the burden of hardware developers.

A program block in the source code can potentially run on multiple DSAs on a DPU. Progressive lowering attempts to generate all possible hardware IRs for each program block. These IRs are semantically equivalent, but have different characteristics in performance, efficiency, and resource cost. In the second phase, NutCracker performs *cost-guided mapping optimization* to derive an optimal implementation from all possible program candidates. To perform this optimization effectively, NutCracker applies equality saturation [51], which encodes all candidate implementations into a graph, with nodes representing semantically equivalent hardware IRs. NutCracker then defines a *joint cost model* that accurately reflects the computation and communication cost of running each candidate implementation on the respective DSA. Based on the cost model, NutCracker's equality saturation selects a hardware mapping with the lowest cost from the graph. Lastly, NutCracker performs code generation to produce the final backend code.

We have implemented eight DPU offloading applications in target-independent high-level languages, and compiled them to native code running on NVIDIA BlueField-2 and BlueField-3 DPUs using NutCracker. Compared to carefully hand-crafted versions of the applications, NutCracker reduces developer effort by 80-93% in lines of code. Moreover, for seven of the eight evaluated applications, the NutCracker-generated code achieves peak throughput within 3% of the hand-crafted baselines. It also outperforms Alkali [25] by up to 47.3×. Furthermore, NutCracker requires zero code changes to port between the two DPUs. This combination of a substantial productivity gain with only a negligible performance trade-off highlights that NutCracker enables high-level, hardware-agnostic programming with optimized hardware performance.

2 Background and Motivation

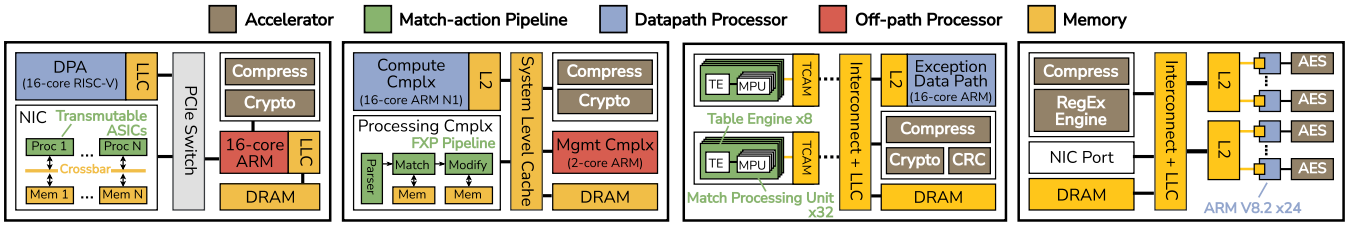
2.1 Hybrid DPUs with Heterogeneous DSAs

Modern SoC-based DPUs comprise heterogeneous DSAs to accelerate datapath processing (as shown in Figure 1). These DSAs fall into three categories distinguished by their control flow, computation, and memory models:

Fixed-function accelerators provide hardware-level specialization through dedicated logic, trading programmability for efficiency. They have static control flow with no support for branching or dynamic execution, and operate on pre-allocated buffers with fixed memory access patterns. These accelerators can be triggered directly from the packet processing path, avoiding context switching and cache pollution. On NVIDIA BlueField-2 [31], SHA256 hashing of a 1KB block completes in 24μs, regex matching on a 20-byte string in 10μs, and compressing a 4MB block in 9ms.

Match-action pipelines adopt a table-driven model, where each pipeline stage matches on packet fields and executes corresponding actions. They support stateful operations (e.g., meters, counters) and basic packet modifications. The memory model differs by architecture: Reconfigurable Match-Action Tables (RMT) [7] use distributed per-stage memory, as in Match Processing Units on AMD Pensando [2, 3] and Packet Processing Engines on Intel IPU [12]. Disaggregated RMT (dRMT) [11] centralizes memory to support efficient cross-stage sharing, as in network ASICs on NVIDIA BlueField [31, 32] and Spectrum [33].

Datapath processors are wimpy general-purpose cores embedded in the packet processing path, executing one packet at a time under a run-to-completion (RTC) [60] model. They support full control flow (e.g., conditionals, bounded loops), basic arithmetic, and flexible memory access within thread-local or shared regions. An example is the Datapath Accelerators (DPAs) in NVIDIA BlueField-3 [34], featuring 16 RISC-V cores and 256 threads for parallelized packet processing.



(a) NVIDIA BlueField-3 [32] **(b)** Intel IPU E2000 [12] **(c)** AMD Pensando [3] **(d)** Marvell OCTEON TX2 [30]

Figure 1. Overview of typical DPU architectures.

However, DPAs face programmability limits: they lack dynamic thread management and inter-thread synchronization, making them unsuitable for complex control or memory patterns. In such cases, ARM cores serve as fallbacks, effectively acting as datapath processors.

2.2 Barriers to Efficient Offloading on Hybrid DPUs

Developers face significant programming burdens when offloading applications to hybrid DPUs with heterogeneous DSAs. These burdens arise from extensive manual effort in partitioning, mapping, and refactoring programs to accommodate the diverse architectures and programming models of different DSAs. We identify two primary barriers:

```
/* P4 */
control Ingress(inout headers hdr) {
    apply {
        hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
    }
}
```

(a)

```
/* C */
void decrement_ttl(struct iphdr *iph) {
    iph->ttl = iph->ttl - 1;
}
```

(b)

```
/* DOCA Flow */
struct doca_flow_actions action = {0};
struct doca_flow_action_descs desc = {0};

desc.type = DOCA_FLOW_ACTION_ADD;
desc.field_op.dst.field_string = "outer.ipv4.ttl";
desc.field_op.dst.bit_offset = 0;
desc.field_op.width = 8;
action.outer.ip4.ttl = 0xFF;

doca_flow_pipe_cfg_set_actions(
        pipe, &action, 0, &desc, 1
);
```

(c)

Figure 2. TTL decrement implementation on (a) P4 MPU in P4, (b) ARM cores in C, and (c) BlueField-3 network ASICs using DOCA Flow APIs.

Barrier#1: Heterogeneous vendor-specific interfaces. Hybrid DPUs expose their DSAs through distinct vendor-specific interfaces, each requiring developers to manually adapt program logic. These interfaces vary not only in API syntax but also in programming semantics, supported data structures, and compilation toolchains. As a result, implementations are often non-portable and tightly coupled to specific hardware. For example, a simple TTL decrement may require three different implementations (Figure 2), ranging from 3 to 10 lines of code, highlighting the engineering overhead imposed by these heterogeneous interfaces.

Barrier#2: Suboptimal mappings within limited space. While heterogeneous DSAs offer significant opportunities, current strategies often confine developers to a limited subset of the implementation space. As shown in Figure 3, different DSAs exhibit significant performance variation even for identical operations, offering a vast implementation space. To

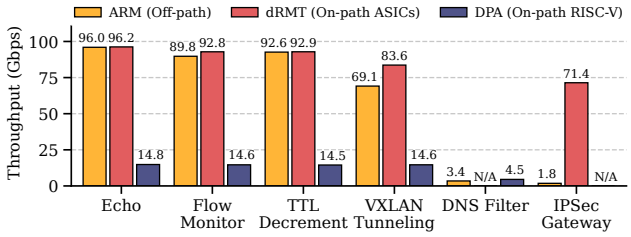


Figure 3. Performance comparison of network functions across programmable components on BlueField-3 [32].

leverage heterogeneous DSAs, developers must partition applications into logical components and map each to a suitable DSA, balancing performance with architectural constraints. However, existing partitioning and mapping strategies are largely driven by hardware capabilities and resource limits. This restricts exploration to a narrow implementation space and overlooks alternatives that could yield better performance. Figure 4 illustrates this challenge with a three-stage network function chain offloaded to BlueField-3. The classification and load balancing stages can be directly translated into DOCA Flow rules and offloaded to network ASICs. In contrast, the sliding window weighted average involves multiplications, which are natively unsupported on ASICs. Although this operation can be rewritten using shift and addition, such transformations require semantic insight and are difficult to discover manually. The challenge intensifies especially as program complexity grows and hardware constraints diverge across DPU generations.

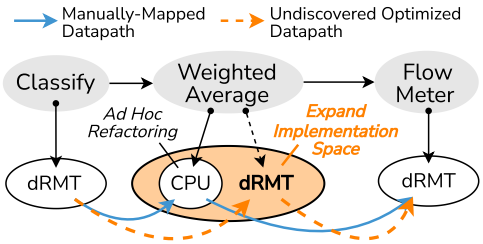


Figure 4. Suboptimal mapping example.

2.3 Limitations of Existing Approaches

Existing programming frameworks fail to cope with the heterogeneity of DPU due to the following reasons:

Homogeneous hardware substrate. Most prior frameworks are designed for a single type of compute resource, such as multicore general-purpose processors or programmable ASICs. The Portable NIC Architecture (PNA) [36] in P4 prioritizes portability across different DPUs, often forfeiting access to DPU-specific accelerations. Floem [37] and iPipe [26] are built for on-path multicore DPUs, specifically the Marvell LiquidIO [29] series, which rely primarily on arrays of ARM cores for packet processing. Alkali [25] targets homogeneous multicore architectures on different DPUs. Lyra [14] supports multiple generations of programmable packet processing ASICs on P4 switches. These frameworks assume a fixed, homogeneous hardware substrate and do not account for the architectural diversity in hybrid DPUs.

Partial use of program optimization. Several frameworks adopt program optimization techniques in compilation, but they primarily focus on resource allocation or partitioning, rather than general-purpose, architecture-aware rewriting. Gallium [57] uses a label-removing algorithm to partition programs between the host CPU and the P4 switches. CaT [16] and Chipmunk [15] apply program synthesis approaches like SMT solver and SKETCH for optimizing resource allocation under target-specific constraints. Alkali [25] leverages an SMT solver and hardware specification constraints to maximize parallelism and resource utilization on multicore DPUs. Lakeroad [46] and Churchroad [45] use synthesis and equality saturation to reason about mapping logic onto FPGA resources. These techniques are effective within fixed hardware models but do not extend to cross-DSA or architecture-aware transformation on hybrid DPUs.

3 NutCracker Overview

We aim to build a compilation framework for hybrid DPUs with the following goals. First, the compiler should enable hardware-agnostic programmability across diverse DSAs, allowing developers to write code without deep knowledge of the underlying hardware. Second, it should optimize program mappings to hardware resources to meet developer-defined objectives, such as maximizing throughput or minimizing energy consumption.

3.1 Design Challenges

C1: Modeling heterogeneous DSA semantics. It is nontrivial to define a unified execution abstraction across diverse DSAs, due to fundamental differences in their control flow structures, computational capabilities, and memory access models. Even within the same architectural class (e.g., pipeline-based packet processors), DSAs can exhibit significant variability caused by vendor-specific constraints and implementation details. For example, NVIDIA BlueField-3 adopts a dRMT-style pipeline with restricted stateful operations, whereas the RMT-style pipeline of Intel IPU enforces fixed-function stage roles.

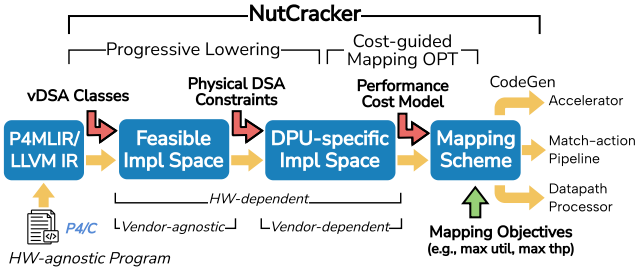


Figure 5. NutCracker workflow overview.

C2: Limited exploration in implementation space. The vast implementation space of hybrid DPUs is rarely explored exhaustively. In practice, developers tend to rely on experience-driven mapping, which only covers a small fraction of the possible implementations. This not only leads to suboptimal performance but also undermines the benefits of heterogeneous DSAs. For instance, a developer may realize a flow classification on the programmable pipeline simply out of familiarity, without considering hardware accelerators that could improve throughput or reduce energy consumption.

C3: Identifying optimal mapping for diverse objectives. Even within the limited space described in C2, it remains challenging to identify program-to-DSA mappings that satisfy developer-specified objectives. The mapping decisions for different objectives (e.g., maximizing throughput or minimizing energy consumption) require reasoning over both intra-DSA computation cost and inter-DSA communication overhead. This complexity is further amplified when scaling to the full implementation space.

3.2 NutCracker Solution

To address the unique demands of hybrid DPUs, we present NutCracker, a compilation framework that enables hardware-agnostic programming with optimal placement. NutCracker is built atop MLIR [22], a modern compiler infrastructure widely used in both academia and industry. MLIR models all program elements as *operations*, ranging from computation to control flow statements, and groups operations into *dialects*, to provide modular, domain-specific abstractions. In response to the challenges outlined in §3.1, NutCracker incorporates the following key designs.

Virtual DSA (§4). In response to **C1**, NutCracker introduces *virtual DSA* (vDSA), a class-based intermediate abstraction serving as MLIR dialects. By adopting a two-layer structure, vDSA captures both common architectural patterns and vendor-specific constraints. At the top layer, *core vDSA classes* model fundamental DSA architectures, such as fixed-function accelerators, match-action pipelines, and data path processors. At the bottom layer, *vendor-specific vDSA instances* instantiate these classes to reflect concrete hardware implementations. This design ensures precise encoding of architectural behaviors while maintaining extensibility to accommodate diverse DSA requirements.

Progressive lowering (§5). To address **C2**, NutCracker employs a *progressive lowering* approach that systematically expands the implementation space while preserving program semantics. We apply rule-based rewriting at both syntactic and semantic levels to generate diverse program variants aligned with vDSAs. By exploring equivalences in control flow, memory access, and computation, NutCracker maximizes optimization opportunities across heterogeneous DSAs. Invalid mappings are pruned based on vendor-specific constraints, ensuring that the final implementation space is both extensive and valid for physical DSAs.

Cost-guided mapping optimization (§6). To tackle **C3**, NutCracker leverages equality saturation (EqSat) [51] to derive an optimal mapping. EqSat encodes the implementation space as a graph of equivalent programs, enabling efficient, exhaustive exploration across mapping alternatives. To tailor EqSat for the hybrid DPU with heterogeneous DSAs, NutCracker introduces a joint cost model that considers both intra-DSA computation cost and inter-DSA communication cost. This model supports objective-aware optimization through cost weighting, allowing developers to prioritize different objectives such as maximizing throughput or minimizing energy consumption.

NutCracker workflow. To this end, Figure 5 shows the complete workflow of NutCracker by putting all key designs together. Developers write hardware-agnostic programs in P4 or C, which NutCracker compiles into an intermediate representation (IR). In addition to the IR, NutCracker accepts a specification of the target hybrid DPU architecture, including the set of available DSAs, their corresponding virtual abstractions, and an associated performance cost model. NutCracker generates a rich implementation space by progressively transforming the hardware-agnostic IR into semantically equivalent variants, each targeting different DSAs. It then performs cost-guided selection to identify a final mapping that aligns with developer-specified objectives, such as maximizing throughput.

4 Virtual Domain Specific Architectures

NutCracker introduces virtual DSAs (vDSAs), a class-based intermediate abstraction that facilitates compilation from hardware-agnostic programs to hybrid DPUs. As shown in Figure 6, vDSA adopts a two-layer structure to capture both common architectural patterns and vendor-specific hardware constraints:

(1) *Core vDSA classes*, which capture common architectural features of DSA, including fixed-function accelerators, match-action pipelines, and datapath processors;
(2) *Vendor-specific vDSA instances*, which instantiate each core class to encode practical hardware constraints observed in commercial DPUs.

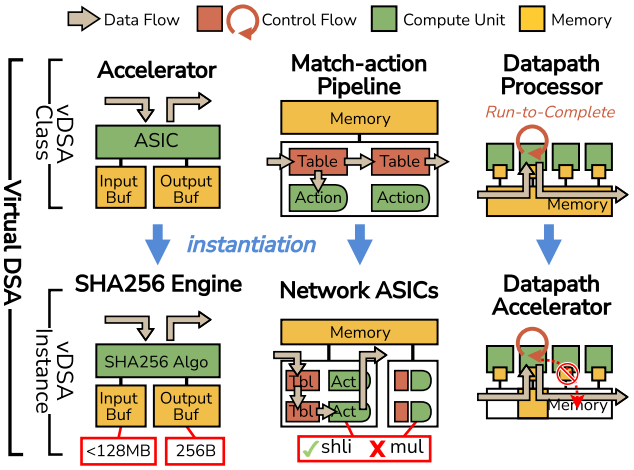


Figure 6. Two-layer structure of vDSA. The core class (top) defines the abstract execution model, while the vendor instance (bottom) enforces physical constraints. Red outlines denote hardware-specific limits, such as buffer capacity (`<128MB`), instruction set restrictions (e.g., lack of `mul`), and restrained memory access.

Each vDSA class is defined as an independent MLIR dialect. Below, we introduce three core vDSA classes for predominant architectural styles found on modern DPUs and illustrate each with representative vendor-specific instances.

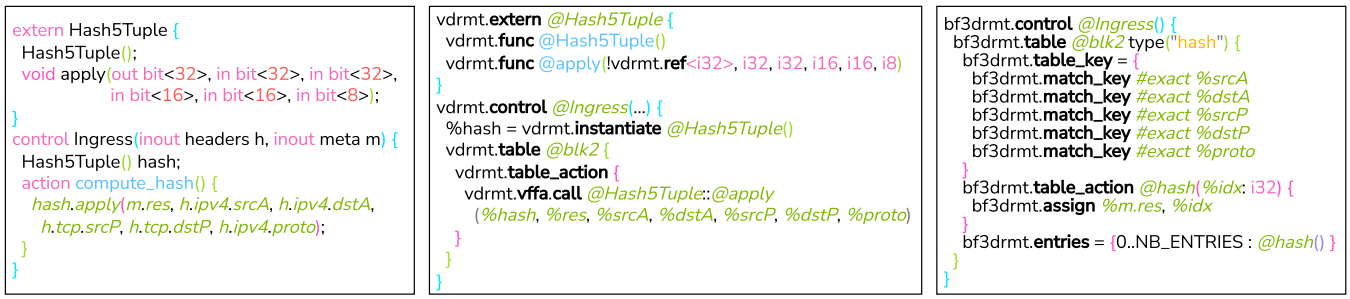
4.1 Fixed-function Accelerator vDSA

The vDSA class of fixed-function accelerators abstracts stateless and atomic operations exposed as external function calls. These operations appear as method-like invocations in frontend languages such as P4 or C, which are treated as opaque compute units during compilation. To reflect hardware-level constraints, this abstraction prohibits internal control flow or memory access, since such accelerators rely on external orchestration and operate over externally managed buffers.

Example: SHA-256 accelerator on Bluefield. We apply this vDSA class to instantiate hardware-accelerated SHA-256 hash calculation on Bluefield. SHA-256 is one of the most widely used hash algorithms (e.g., digital certificates, flow classification, and load balancing). It always produces a 256-bit value, regardless of the input data size. However, Bluefield-2 enforces concrete hardware constraints, i.e., the SHA engine accepts input up to 128 MB [4]. This instance conforms to the accelerator execution model without internal state or control flow, treated as a pure external function.

4.2 Match-action pipeline vDSA

The vDSA class of match-action pipelines models pipeline-based packet processing through a sequence of control flow, computation, and stateful operations. Control-flow operations support diverse pattern-based matching (e.g., exact match, ternary match) followed by action selection, reflecting the key semantics of match-action stages. Computation



(a) Hardware-agnostic P4 program using a fixed-function accelerator (FFA) for five-tuple hashing

(b) Core vDSA class: vDRMT interacting with a virtual FFA (vFFA) via function call

(c) Vendor-specific vDSA instance: Mapping the vFFA to a hardware hash table on BlueField-3

Figure 7. A five-tuple hash in (a) hardware-agnostic P4, (b) core vDSA (`vDRMT`), and (c) vDSA instance (`bf3drmt`).

operations support header manipulation and simple arithmetic. Stateful operations enable access to registers and counters, typically expressed through external function calls. This abstraction captures the sequential table-driven execution model of programmable match-action pipelines and generalizes across both RMT and dRMT hardware.

Example: Network ASICs on BlueField. We instantiate the match-action pipeline vDSA to model the dRMT-style pipeline of BlueField-3, imposing various vendor-specific constraints. These constraints are encoded into the vDSA instance, enforcing strict boundaries compared to the general match-action pipeline model. For computation operations, this instance restricts supported arithmetic operations by omitting hardware multiplication and division. For stateful operations, it limits memory access to simple primitives such as counters and meters.

4.3 Datapath processor vDSA

The vDSA class of datapath processor abstracts the RTC-based packet processing model. It avoids the stringent timing constraints in the pipeline and hence enables flexible programmability. This class supports (i) control-flow operations such as conditional branches and bounded loops, (ii) computation operations for basic integer arithmetic, and (iii) flexible memory accessing operations to a shared memory pool. The RTC-based model relies on hardware-based scheduling, lacking support for dynamic thread management.

Example: DPAs on BlueField-3. We instantiate the datapath processor vDSA to reflect the execution model and memory constraints of RTC-based DPA cores on BlueField-3. This instance enforces concrete hardware limitations. On the one hand, this instance does not support floating-point arithmetic. On the other hand, its memory access is restricted to statically allocated and thread-local memory regions to ensure memory safety and isolation.

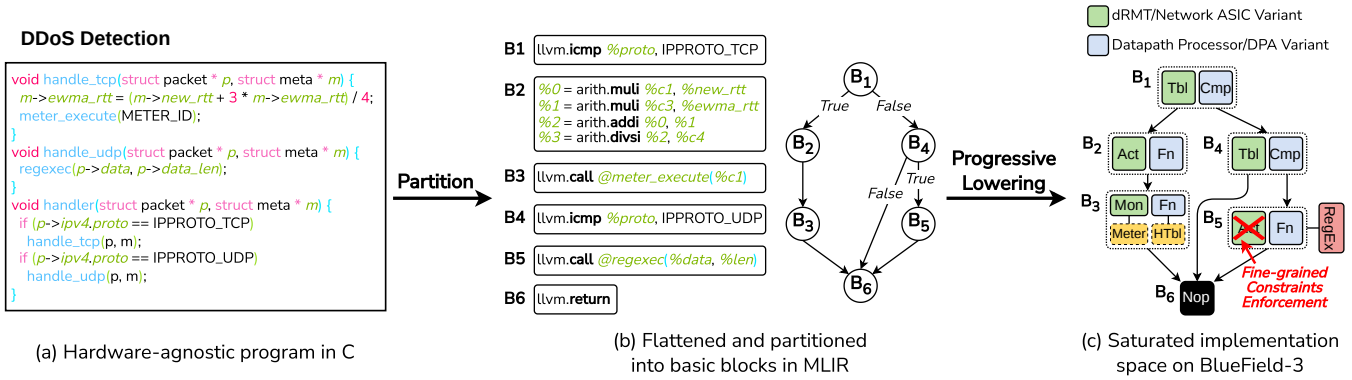
4.4 vDSA in Practice: Five-Tuple Hashing

With the two-layer vDSA design, NutCracker exposes two distinct interfaces: one for application developers and one for hardware vendors. For developers, NutCracker provides standard headers in frontend languages (e.g., P4 or C) that declare fixed-function accelerators as abstract `extern` objects. During compilation, invocations of these `extern` objects are transformed to function calls to virtual fixed-function accelerators (vFFAs). Notably, vFFAs are not encapsulated in a standalone MLIR dialect. Instead, they are embedded into macro vDSA dialects, such as virtual disaggregated match-action tables (`vDRMT`) or datapath processors (`vDPP`). For vendors, NutCracker allows them to specialize the core vDSA dialect into custom MLIR dialects that model hardware-dependent physical instances. Specifically, vendors enforce hardware constraints via MLIR operation verifiers (e.g., operand type checks) and encode conversions between MLIR operations in the form of rewrite rules.

We illustrate these two interfaces using *five-tuple hashing* as an example. The five-tuple hashing is a common functionality in network applications, used in tasks such as Receive-Side Scaling (RSS) and load balancing. As shown in Figure 7(a), the developer instantiates the `Hash5Tuple` and invokes its `apply` method in a hardware-agnostic P4 program. In the core `vDRMT` dialect (Figure 7(b)), the `apply` method is expressed as a function call to a virtual hash accelerator via the standardized `vdrmt.vffa.call` operation. At this stage, the compiler remains agnostic about whether this accelerator will ultimately map to a physical hardware engine or a software fallback. When targeting the BlueField-3 Network ASIC, NutCracker maps this virtual hash accelerator to a specialized match–action table on the ASIC. This table uses the packet five-tuple as its key and relies on the ASIC's built-in hash function to compute the result implicitly. NutCracker models this construct in the `bf3drmt` dialect as a `bf3drmt.table` operation annotated with `type("hash")`. In this way, NutCracker realizes a semantically equivalent five-tuple hashing on Network ASICs while satisfying the target-specific constraints of BlueField-3.

5 Progressive Lowering

Overview. NutCracker introduces a progressive lowering approach to construct a valid *DPU-specific implementation*



(a) Hardware-agnostic program in C

(b) Flattened and partitioned into basic blocks in MLIR

(c) Saturated implementation space on BlueField-3

Figure 8. End-to-end example of offloading DDoS Detection to NVIDIA BlueField-3 DPU (two candidate vDSAs: dRMT for the NIC network ASICs and datapath processors for DPA cores and ARM cores)

space from an input hardware-agnostic program. In short, it first explores semantically equivalent program implementations across virtual DSAs and then prunes invalid ones based on vendor-specific constraints of physical DSAs. The core of progressive lowering is ***rule-based rewriting***, a technique widely used in programming languages and compilation [53, 58] to systematically transform program terms while preserving original semantics.

Design principle. The key principle of progressive lowering is to maximize optimization opportunities across heterogeneous DSAs. In contrast to traditional compilers that prematurely eliminate DSA candidates based on hardware capacity, NutCracker classifies operations and explores the implementation space at both syntactic and semantic levels. For control-flow and memory-accessing operations, we apply syntactic-level rewriting to generate equivalent block variants aligned with diverse vDSA classes. For computation operations, we apply semantic-level rewriting to further identify deep equivalence among expressions, retaining more valid block variants for mapping. By combining these two stages, NutCracker expands the number of feasible implementations, thereby increasing the likelihood of generating valid physical DSA mappings.

Workflow. The progressive lowering approach proceeds in three compilation stages. Its input includes (1) a hardware-agnostic input program written in a frontend MLIR dialect, e.g., P4MLIR or LLVM IR; (2) the candidate physical DSAs on the target DPU; and (3) their associated vDSA classes. The output is a set of target-specific implementations in physical DSA MLIR dialects. Specifically, the workflow is as follows:

I. **Program partitioning (§5.1).** This stage decomposes the program into basic blocks, isolating operations that are sensitive to hardware-specific behaviors to enable flexible mapping across heterogeneous DSAs.

II. **Coarse-grained Rewriting (§5.2).** This stage performs *syntactic-level* rewritings to align each block with compatible vDSA classes, generating a feasible implementation space.

III. **Fine-grained Rewriting (§5.3).** This stage applies both syntactic- and *semantic-level* rewritings to enforce vendor-specific constraints, narrowing the feasible implementation space to variants compatible with physical DSAs on the target DPU.

Example. Figure 8 shows the complete progressive lowering workflow on a DDoS detection application. The program is first partitioned into six basic blocks (Figure 8(a–b)): where B1 and B4 perform protocol checks (TCP and UDP), B2 and B3 realize stateless metadata processing and metering (`handle_tcp`), B5 executes regex matching (`handle_udp`), and B6 returns. During coarse-grained rewriting (Figure 8(b–c)), each block is lowered into two variants. For example, conditional branches (`llvm.icmp`) is lowered to `drmt.match_key` of dRMT vDSA and `dpa.cmp` of datapath processor vDSA. Finally, fine-grained rewriting enforces hardware constraints of BlueField-3. Here, B5 cannot be mapped to the dRMT variant since network ASICs on BlueField-3 lack deep packet inspection support (red cross in Figure 8(c)).

5.1 Program Partitioning

NutCracker begins by decomposing the program into *basic blocks*, enabling flexible mapping across heterogeneous DSAs. Similar to conventional compilers, a basic block is a straight-line sequence of operations with a single entry and exit point. The major difference is that NutCracker imposes two additional constraints to isolate operations sensitive to DSA-specific control and memory behavior:

(1) Each basic block contains *at most one* control flow operation (e.g., table matches, conditional branches);

(2) Each basic block contains memory-accessing operations that target the *same memory position* (e.g., register writes or counter updates to the same index).

These constraints ensure that each block remains independently analyzable and mappable to DSAs with varying capabilities and semantics. NutCracker applies two key transformations to achieve this partitioning: *control flow flattening* and *memory accessing isolation*.

Control flow flattening. To enforce the first constraint, NutCracker transforms control flow operations into structured subgraphs of basic blocks that separate predicate evaluation from execution paths. The main control flow operations flattened are *function calls* and *table invocations*, which are expanded into basic block subgraphs that explicitly enumerate all possible execution paths. External function calls are preserved because they represent interactions with accelerators and will not be expanded. This transformation is essential because different DSAs expose distinct control models: some (e.g., network ASICs) rely on match-action tables while others (e.g., datapath processors) offer imperative branching. Isolating control logic simplifies downstream mapping to vDSAs with such difference.

Memory accessing isolation. To enforce the second constraint, NutCracker ensures that all memory-accessing operations targeting the same memory position are placed within the same basic block. This transformation is necessary because DSAs typically operate over isolated memory regions and lack support for disaggregated memory accesses across DSAs. If this constraint is not enforced, blocks that access the same memory object may be mapped to separate DSAs, violating hardware isolation guarantees and leading to incorrect execution, such as inconsistent state updates. Specifically, the transformation consists of two steps:

First, NutCracker partitions the program so that each block contains *at most one* memory-accessing operation (e.g., a register or counter read/write). Stateless computations that are independent of memory operations, such as packet header modifications, are placed in separate blocks. Computations that are required by memory operations (e.g., index or key calculations) are kept within the same block.

Second, NutCracker merges all memory-accessing operations that target the same memory position, along with any intermediate blocks, into a single basic block. This ensures that related memory-accessing operations remains colocated and can be safely mapped to DSAs with constrained memory models.

5.2 Coarse-grained Rewriting

NutCracker applies coarse-grained rewriting to transform each basic block into semantically equivalent variants aligned with one or more candidate vDSA classes. This stage takes the basic blocks partitioned from the hardware-agnostic program and produces a *feasible implementation space*, a set of basic block variants expressed in vDSA dialects. At this stage, the rule-based rewriting targets operations that are most sensitive to DSA differences: control flow operations and memory-accessing operations. Thus, the obtained variants comply to the control and memory models of candidate vDSA classes but remain agnostic to vendor-specific constraints (e.g., lack of floating point). That is to say, they are hardware-dependent but still portable across different vDSA instances that belong to the same vDSA class.

Control flow operations are mapped to match-action tables on match-action pipeline vDSA (e.g., RMT), or to conditional branching and hash-table lookups on datapath processor vDSA (e.g., DPA). After applying control flow flattening (§5.1) to eliminate function calls and table applications, NutCracker rewrites basic blocks containing the following control flow operations:

(1) **Bounded loops** are unrolled into straight-line sequences of basic blocks. This unrolling is implemented as an *MLIR pass*, a pattern-based rewriting mechanism used in transformation. Currently, we only support loops with static bounds, as unbounded loops are typically unsupported by DSAs due to non-termination risks.

(2) **Conditional branches** are rewritten into exact-match table entries for match-action pipeline vDSAs. These rewritings are expressed declaratively in DialEgg [56], a framework enabling dialect-agnostic rewriting over MLIR input. For instance, an equality comparison branch in P4 can be rewritten as an entry within a match-action table in the dRMT vDSA.

(3) **Match operations**, including exact match, ternary match, and longest prefix match (LPM), are rewritten to appropriate data structures in datapath processor vDSAs. Specifically, exact matches are transformed into conditional branches, ternary matches into hash-table lookups with masking, and LPMs into software-based LPM table queries. Similarly, these transformations are also declared in DialEgg.

Memory-accessing operations are mapped to local or shared memory, conforming to the memory constraints (e.g., locality and sharing pattern) of each vDSA. NutCracker classifies memory objects based on their access locality (within a block or across blocks) and whether their access patterns can be partitioned (e.g., by packet header fields), to simplify alignment with vDSA memory models. NutCracker applies one of three possible rewrites based on the property check:

- *Local memory object*: If a memory object is accessed only within a single basic block, it is rewritten as a local instance. This maps to a local table in RMT, a shared table in dRMT, or a per-thread hash table in a datapath processor.
- *Shared memory object* with *partitionable* key space: If a shared memory object is indexed using a packet-derived key (e.g., 5-tuple hash), NutCracker partitions it on vDSAs that do not support shared memory, and preserves it as shared on those that do. On RMT vDSA, NutCracker inserts a compiler-generated match-action table that matches on the access key and dispatches packets to replicated basic blocks. Each block contains a local table for a disjoint partition of the key space. On datapath processor vDSA, the key is used to partition hash tables across processing cores, with each core responsible for the table in its per-thread memory. On dRMT vDSA, the memory instance remains globally shared in the disaggregated memory.
- *Shared memory* with *non-partitionable* key space: If the memory object is shared and lacks a usable partitioning key (e.g., a globally shared counter), NutCracker avoids

partitioning it. In such cases, any basic block accessing the memory object cannot be mapped to RMT due to its lack of support for shared memory. dRMT vDSA allows this mapping because it supports globally shared memory via its disaggregated architecture. On datapath processor vDSA, the memory object is either confined to a single core or rerouted to the control path to ensure correctness.

5.3 Fine-grained Rewriting

NutCracker applies fine-grained rewriting to transform the vDSA-aligned variants into vDSA instances on the target DPU. This step narrows the feasible implementation space by enforcing vendor-specific limitations (e.g., restricted arithmetic), generating a DPU-specific implementation space. The obtained variants are in hardware- and vendor-dependent intermediate form in MLIR. This transformation guarantees that each variant in this implementation space is directly compatible with at least one vDSA instance, which can be safely used in the downstream cost-guided mapping process.

At this stage, the rewrite rules primarily target *computation operations* (e.g., arithmetic and bitwise) that may be constrained by the hardware capabilities of physical DSAs. These rewrite rules are handcrafted based on known low-level transformation strategies. For example, multiplication may be rewritten using shifts and additions, while certain bitwise operations can be expressed through concatenation and vector extraction. NutCracker currently includes 3 rules for transforming multiplication and division, along with 14 rules for bitwise operations, specifically designed to rewrite variants in dRMT vDSAs into compatible variants for BlueField-3 network ASICs. Nevertheless, this handful of rules is sufficient to support a wide range of common computations in network functions, including header field extraction through bit shifting and implementation of custom hashing logic.

Figure 9 illustrates how NutCracker rewrites a bitwise `AND` operation into a semantically equivalent expression using concatenation, when the target DSA lacks native support for bitwise operations. The initial e-graph (left) represents the expression a & 0xF0, where a is an 8-bit bitvector. a_{hi} and a_{lo} denote the high and low 4-bit slices of a respectively, and the $+\!\!+$ operator denotes bitvector concatenation. Several rewrite rules are applied to this expression, including: 1. $a \Rightarrow a_{\text{hi}} +\!\!+ a_{\text{lo}}$, 2. $(a_{\text{hi}} +\!\!+ a_{\text{lo}}) \& (b_{\text{hi}} +\!\!+ b_{\text{lo}}) \Rightarrow (a_{\text{hi}} \& b_{\text{hi}}) +\!\!+ (a_{\text{lo}} \& b_{\text{lo}})$, 3. a & 0x$F \Rightarrow a$, and 4. a & 0x0 $\Rightarrow$ 0. Applying these rewrites merges the expression $a_{\text{hi}} +\!\!+$ 0x0 into the same e-class as the original term a & 0xF0.

6 Cost-guided Mapping Optimization

Overview. NutCracker selects efficient mappings by applying *equality saturation* (EqSat) [51], which enables exhaustive exploration of the DPU-specific implementation space. Specifically, it encodes this space as an *e-graph* composed

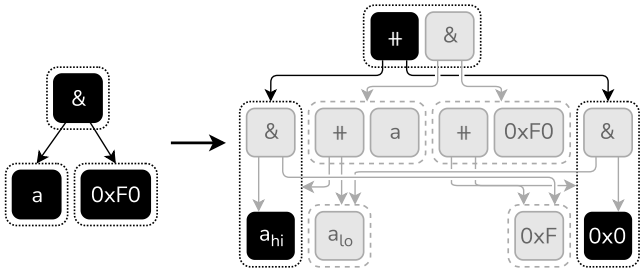


Figure 9. Example of e-graph before and after saturation. The dotted boxes show e-classes and the solid boxes show e-nodes. The black e-nodes are the effective operands.

of multiple *e-classes*, where each class contains *e-nodes* representing semantically equivalent variants of a basic block mapped to different DSAs. NutCracker then applies a joint cost model (§6.1) to estimate the overheads across all candidates. It extracts the lowest-cost mapping from the e-graph by recursively traversing from the root and selecting optimal e-nodes. This process is guided by developer-specific compilation modes (§6.2).

6.1 Joint Cost Model on E-graph

Limitations. Conventional EqSat frameworks face two key limitations when applied to heterogeneous DPU mapping. First, they assume that only computation nodes incur cost, treating edges merely as structural dependencies with zero cost. This abstraction fails in hybrid DPUs, where edges may represent actual communication between different DSAs, introducing non-negligible overhead and hardware-specific feasibility constraints. Second, existing cost-guided extraction strategies typically rely on precise performance models for each backend. In practice, such detailed performance data is often unavailable, as vendors seldom release fine-grained cost models or standardized benchmarks across DSAs.

Joint cost model. To address these limitations, NutCracker introduces a joint cost model that works with *relative cost estimation*. So far, building a cycle-accurate model remains impractical for two reasons. First, vendors do not disclose detailed performance specifications for their DSAs, treating them as proprietary IP. Second, many DSAs lack hardware counters that would support direct per-operation profiling. To address the challenge, we construct *virtual cost* values for operations, inferred from empirical profiling, and use their normalized ratios to guide the mapping decisions. Our joint model integrates three costs: 1. *computation cost* of executing each basic block on a DSA, 2. *energy cost* of activating a DSA, and 3. *communication cost* incurred when transferring data between blocks mapped to different DSAs.

Intra-DSA computation cost. The virtual computation cost models the relative execution speed of different DSAs. For example, (i) accelerators incur low computation cost due to fixed-function efficiency, (ii) match-action pipelines incur

moderate cost, and (iii) datapath processors and general-purpose cores incur higher costs. The computation cost of a basic block comes from two sources:

- **Independent operations** (e.g., arithmetic on thread-local data) do not involve synchronization or shared resources dependencies. These operations scale linearly with the available compute units due to their inherent parallelism.
- **Shared resource operations** (e.g., accesses to shared hash tables) incur coordination overhead due to contention. As the number of compute units grows, their performance scales sub-linearly.

Given a mapping where basic block b executes on DSA d with parallelism degree P, we model its computation latency and throughput as follows:

$$\text{Cost}_{\text{indep}} = \sum_{op \in \text{Ops}_{\text{indep}}} \text{cost}_d(op), \quad \text{Cost}_{\text{shared}} = \sum_{op \in \text{Ops}_{\text{shared}}} \text{cost}_d(op)$$

$$\mathcal{L}_{\text{Comp}} = \text{Cost}_{\text{indep}} + \text{Cost}_{\text{shared}} \cdot (1 + \sigma_d(P-1)), \quad \mathcal{T}_{\text{Comp}} = \frac{P}{\mathcal{L}_{\text{Comp}}}$$

Here, $\text{Cost}_{\text{indep}}$ and $\text{Cost}_{\text{shared}}$ represent the base execution latency of the block's operations on a single compute unit of DSA d, by summing the latency $\text{cost}_d(op)$ of all operations within the corresponding sets. To estimate the total computation latency $\mathcal{L}_{\text{Comp}}$ across P parallel units, we combine the constant independent latency with the shared-state latency, applying a synchronization penalty modeled by $\sigma_d(P-1)$. Here, $\sigma_d \in [0, 1]$ parameterizes the contention overhead specific to the hardware architecture of d. To estimate the computation throughput $\mathcal{T}_{\text{Comp}}$, we divide the allocated parallelism P by this total computation latency, demonstrating how throughput scaling is explicitly bounded by the increased latency of shared operations in the denominator.

Intra-DSA computation cost enables us to differentiate the scaling characteristics of diverse DSAs. For instance, Network ASICs on BlueField-3 support efficient sharing of stateful meters between hardware pipelines at near line-rate; hence, we set $\sigma_d = 0$ for ASIC meter operations, allowing throughput to scale perfectly with P. In contrast, on ARM cores, metering relies on software structures like hash tables. Concurrent accesses to these structures incur lock contention and cache-coherence traffic, severely limiting scalability. Therefore, we assign $\sigma_d = 1$ to meter operations on ARM cores to reflect this contention penalty.

Intra-DSA energy cost. The virtual energy cost captures the relative power efficiency of executing a block on a specific resource. Since precise energy consumption profiles for the BlueField-3 are not publicly available, we empirically assign energy costs to the onboard components. We assign the lowest energy consumption to the Network ASICs since their domain-specific match-action pipelines process network operations with greater energy efficiency than general-purpose cores. This is followed by the DPA cores, which operate at a lower clock frequency, while the general-purpose ARM cores are assigned the highest relative energy cost.

Inter-DSA communication cost. The virtual communication cost is defined according to the known on-die layout and interconnect properties. For example, intra-DSA communication between blocks on the same Network ASIC is assigned a low cost due to its low-latency internal bus. In contrast, cross-DSA communication between the Network ASIC and the ARM core is assigned a much higher cost, as it involves traversing a PCIe switch with higher latency. In cases where no viable communication path is known or officially supported, such as from DPA to ARM cores, NutCracker assigns an infinite cost to the corresponding edge, effectively disallowing such mappings in the optimization phase.

Formally, for a data dependency edge between a source block mapped to DSA d_u and a destination block mapped to d_v, we define the communication latency $\mathcal{L}_{\text{comm}}$ and communication throughput $\mathcal{T}_{\text{comm}}$ as:

$$\mathcal{L}_{\text{comm}}(d_u, d_v) = \begin{cases} 0, & d_u = d_v \\ L(d_u, d_v), & d_u \neq d_v \wedge \text{path supported} \\ \infty, & \text{Otherwise} \end{cases}$$

$$\mathcal{T}_{\text{comm}}(d_u, d_v) = \begin{cases} \infty, & d_u = d_v \\ BW(d_u, d_v), & d_u \neq d_v \wedge \text{path supported} \\ 0, & \text{Otherwise} \end{cases}$$

where $L(d_u, d_v)$ and $BW(d_u, d_v)$ denote the empirical latency penalty and bandwidth limits of the specific hardware interconnect between the two distinct DSAs.

6.2 Compilation Mode for Mapping Objectives

NutCracker offers different compilation modes to meet developer-specific mapping objectives. Different scenarios may prioritize different goals, such as maximizing throughput, minimizing latency, or minimizing energy consumption. These goals often conflict with each other, e.g., throughput-oriented mappings may increase power usage, while prioritizing energy savings may result in higher latency.

NutCracker compilation modes. We provide three predefined compilation modes to accommodate common deployment goals: maximizing throughput, minimizing median latency, and minimizing energy consumption. Each mode corresponds to a distinct compilation flag:

- **`-Olatency`** optimizes the end-to-end processing latency by minimizing the critical path. The execution time of a candidate mapping is evaluated by accumulating the computation cost of individual blocks and the communication overhead between the DSAs hosting them along the longest execution path:

$$\text{Cost}_{\text{lat}} = \sum_{b \in \text{Blocks}} \mathcal{L}_{\text{Comp}}(b) + \sum_{(u,v) \in \text{Links}} \mathcal{L}_{\text{comm}}(d_u, d_v)$$

  where Blocks represents the set of all basic blocks in the mapped program, and Links denotes the inter-block communication edges connecting a source block u on DSA d_u and a destination block v on DSA d_v.
- **`-Othroughput`** prioritizes execution throughput, potentially at the expense of increased power consumption. The

Applications	Description	NutCracker LoC	HandCraft LoC
Compression	Packet compression using the **DEFLATE** algorithm to reduce payload size before transmission.	104 (P4)	541
File Verification	Integrity checking using **SHA-256** hashing over packet contents.	127 (P4)	646
IngressINT	Flow control for IP traffic using a Bloom filter register indexed by **5-tuple hash**.	131 (P4)	1836
IP Router	IPv4 forwarding using **longest prefix match (LPM)** lookup; updates destination MAC address.	119 (P4)	1570
SYN Flood	Detection of TCP SYN flood attacks by tracking SYN, ACK, and SYN-ACK packet counts.	150 (P4)	1670
MazuNAT	Network Address Translation that implements external-to-internal IP mapping for flows.	97 (C)	1326
Flow Meter	**Shared metering** across UDP and TCP traffic; maintains **separated counters** for TCP HTTP and UDP Syslog traffic.	152 (C)	1858
DDoS Detection	**Filters DNS traffic** using a blacklist and **meters TCP traffic** to detect abnormal bursts.	133 (C)	1873

Table 1. Real-world applications used in evaluation.

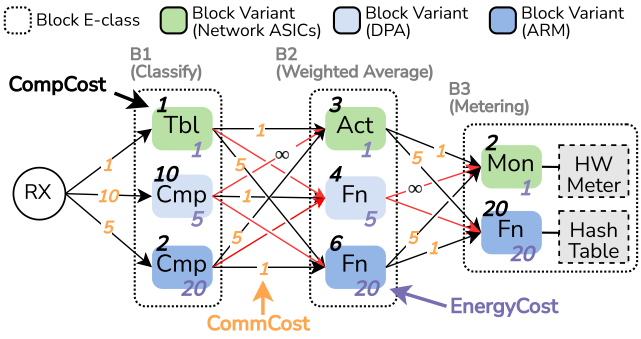


Figure 10. Applying the joint cost model to the TCP path of DDoS Detection, represented as an e-graph.

overall throughput is bottlenecked by the slowest block or communication link. Since standard e-graph extraction is designed to *minimize* a cost, we formulate this objective by minimizing the bottleneck processing time (i.e., the maximum inverse throughput):

$$\text{Cost}_{\text{thp}} = \max\left(\max_{b\in\text{Blocks}}\left(\frac{1}{\mathcal{T}_{\text{Comp}}(b)}\right), \max_{(u,v)\in\text{Links}}\left(\frac{1}{\mathcal{T}_{\text{comm}}(d_u,d_v)}\right)\right)$$

This also strictly prevents invalid mappings: Unsupported communication paths between DSAs contribute to an infinitely high cost, ensuring they are naturally pruned by the cost-minimizing extractor.

- `-Oenergy` minimizes energy consumption by favoring energy-efficient DSA resources, such as the Network ASICs. The compiler accumulates the virtual energy cost $\mathcal{E}_{\text{Comp}}$ for each activated block:

$$\text{Cost}_{\text{energy}} = \sum_{b\in\text{Blocks}} \mathcal{E}_{\text{Comp}}(b)$$

This mode penalizes the use of power-hungry general-purpose cores, maximizing offload to energy-efficient DSAs even at the expense of throughput and latency.

Example. Figure 10 shows the e-graph corresponding to the TCP path in the DDoS Detection program. The e-graph consists of three e-classes, each representing a basic block: classification, timestamp averaging, and flow metering. Each e-class includes multiple e-nodes, representing semantically equivalent variants of the same basic block, each targeting a different DSA on BlueField-3. The traditional EqSat cost model selects one e-node from each e-class and minimizes the sum of the cost on e-nodes, treating edges as cost-free. In this case, EqSat may select a mapping that connects a classification block variant on the DPA to a timestamp averaging block on the ARM core. Although this mapping is logically valid in the original hardware-agnostic program, it is invalid on BlueField-3, where no direct DPA-to-ARM communication path is exposed to developers. NutCracker resolves this limitation by integrating edge costs into the cost model. This allows the compiler to avoid invalid mappings, even when the selected DSAs have low computation costs.

7 Evaluation

Our evaluation aims to answer the following questions:

- How much implementation effort does NutCracker save in developing applications for hybrid DPUs (§7.1)?
- How does NutCracker's performance compare to hand-crafted code and Alkali [25] (§7.2)?
- What is the compilation time of NutCracker compared to existing toolchains (§7.3)?
- How does each stage of progressive lowering impact the final performance of the generated code (§7.4)?
- Can NutCracker identify alternative cross-DSA mappings that expose trade-offs between throughput, energy, and CPU utilization (§7.5)?

Testbed: We use two server–client pairs of Supermicro 2U machines, each connected via direct 100 Gbps Ethernet links. On the server side, one machine is equipped with an NVIDIA BlueField-2 DPU, and the other with a BlueField-3. The client paired with the BlueField-2 server has 64 Intel Xeon Gold 6326 cores (@2.90 GHz) and a 100 Gbps Mellanox ConnectX-5 NIC. The client paired with the BlueField-3 server has 128 Intel Xeon Gold 6548Y+ cores (@4.10 GHz) and a 200G bps Mellanox ConnectX-7 NIC. All clients use an open-loop load generator to send DPDK traffic to the server.

7.1 Implementation Efforts with NutCracker

We implemented eight representative DPU-offloaded applications using NutCracker, as listed in Table 1. The hand-crafted baselines are implemented by a student with BlueField expertise, adapting official NVIDIA DOCA examples into minimal working implementations. Table 1 lists the Lines of Code (LoC) for both the hand-crafted implementations and NutCracker program. Across all applications,

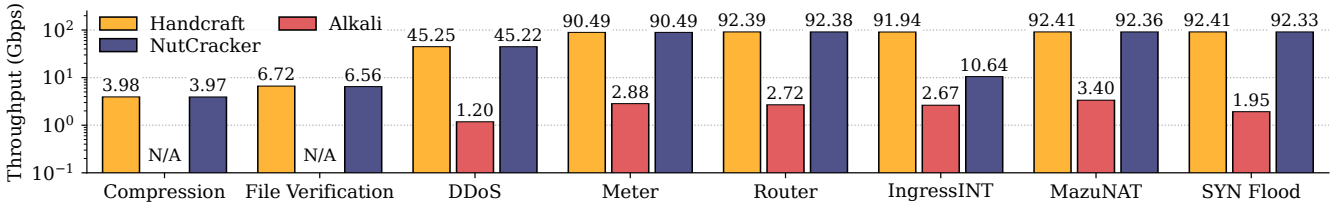


Figure 11. Throughput comparison of NutCracker applications on BlueField-3.

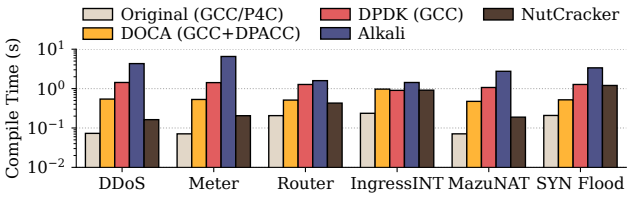


Figure 12. Compilation time of NutCracker compared to other compiler toolchains.

NutCracker reduces LoC by 5–14× compared to the hand-crafted baseline. This reduction is primarily enabled by the vDSA design in NutCracker, decoupling application logic from low-level, DPU-specific APIs. Developers express core logic in hardware-agnostic P4 or C, saving labor in refactoring with vendor-specific APIs.

7.2 Performance on Hybrid DPUs

We evaluate all applications across three implementations: a hand-crafted, manually optimized baseline, NutCracker, and Alkali, a state-of-the-art DPU compilation framework. Specifically, we evaluate the open-source version of Alkali [52] using its default BlueField backend, which targets the ARM cores on BlueField. Figure 11 compares the end-to-end throughput of all applications generated on the BlueField-3 using the `-Othroughput` flag. We use 3100-byte packets for compression, 1200-byte packets for file verification, and 256-byte packets for the other applications.

As shown in Figure 11, NutCracker nearly matches the peak throughput of the hand-crafted baseline and substantially outperforms Alkali. For 256-byte workloads (IP Router, MazuNAT, SYN Flood, and Flow Meter), NutCracker sustains over 90 Gbps, within 2% of the manually optimized implementations (92 Gbps). In contrast, Alkali executes exclusively on ARM cores and does not utilize Network ASICs or DPA accelerators, which limits its achievable throughput on hybrid DPUs. These results indicate that the intermediate code generated by NutCracker does not significantly affect throughput after downstream compilation with GCC or the vendor compiler. Although progressive lowering adds additional logical stages (e.g., extra tables and control-flow transitions), their impact on end-to-end throughput is negligible in our experiments.

The only exception is IngressINT, for which NutCracker reaches 10.27 Gbps, compared to 91.94 Gbps for the hand-crafted baseline. This performance degradation arises from the execution of P4 keyless tables on the CPU. Since keyless tables lack an explicit lookup key, NutCracker lowers them to CPU code that iterates over all table entries at runtime. Although DPDK provides an efficient `rte_hash` implementation for match–action tables that can be shared across cores, the iteration cost depends on runtime configuration, such as table size. These dynamic effects are not captured by our current static cost model, leading NutCracker to underestimate the CPU overhead for IngressINT. We further analyze this limitation in more detail in §7.4.

7.3 Effectiveness of NutCracker Compilation

We measure NutCracker's end-to-end compilation time across six applications and compare it against three baseline toolchains. Figure 12 reports compilation time for four workflows: (1) Original: Hardware-agnostic source code in C or P4, compiled with native toolchains (GCC or P4C). (2) DOCA: C with DOCA APIs, compiled via GCC and DPACC. (3) DPDK: C with DPDK APIs, compiled with GCC (`-O3`). (4) NutCracker: Compiles the hardware-agnostic code into DPU-specific source code via our toolchain, excluding the downstream compilation with the vendor toolchain.

Overall, NutCracker compiles programs on the order of seconds, a timescale comparable to standard industrial toolchains. Furthermore, by avoiding the exhaustive program synthesis, NutCracker achieves a 1.6× to 31.9× compilation speedup compared to Alkali.

App	#Blks	Partition	Lowering	Mapping	Total
IngressINT	13	298.3ms	580.6ms	13ms	912ms
Router	12	54.7ms	342.8ms	11ms	429ms
SYN Flood	13	263.6ms	801.5ms	116ms	1,201ms
NAT	6	<0.1ms	155.4ms	13ms	189ms
Meter	14	4.6ms	168.5ms	12ms	205ms
DDoS	9	1.37ms	131.5ms	10ms	163ms

Table 2. Breakdown of application compilation time.

7.4 Effectiveness of Progressive Lowering

Table 2 further breaks down compilation into partitioning, progressive lowering, and mapping. Across the evaluated workloads, progressive lowering consistently dominates compilation time, accounting for 64–82% of total compilation time. The cost of this phase is primarily driven by two factors: the complexity of the input program and the architectural diversity of the target DPU. As the number of available DSA

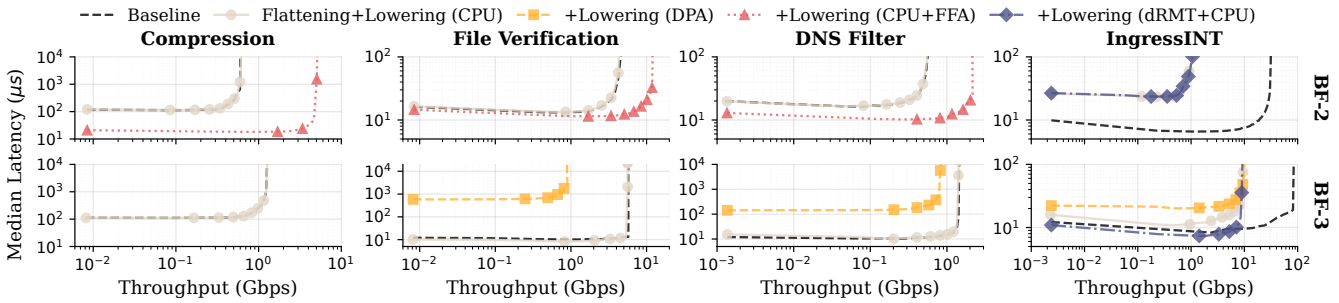


Figure 13. Latency comparison of different stage in progressive lowering.

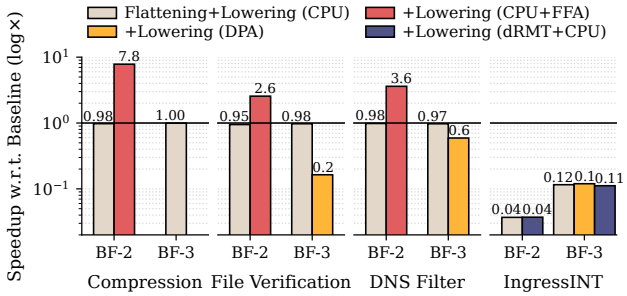


Figure 14. Throughput speedup relative to hand-tuned baselines.

backends increases, the compiler must evaluate a larger set of DSA-specific rewrite rules and mapping options, which increases the time spent exploring the search space.

We use Compression, File Verification, the DNS Filter from DDoS Detection, and IngressINT as case studies to break down the performance contribution of different stages in progressive lowering. Figure 14 and Figure 13 report throughput speedup and median latency for each application at these stages. We normalize throughput against manually optimized baselines implemented by the student with expertise in BlueField-2 (BF-2) and BlueField-3 (BF-3) DPUs.

Minimal control flow flattening overhead. The initial step of flattening control flow and memory operations to the ARM cores (`Flattening+Lowering (CPU)`) incurs minimal overhead and achieves performance close to the hand-crafted baselines for most applications. Although this stage may introduce additional match-action tables on the NIC and extra branching logic on the processors, its impact on the end-to-end performance is rather negligible in our experiments. Specifically, the BlueField ASIC's dRMT architecture comfortably supports ~100 such tables (1K entries each) without significantly degrading throughput or latency. Because NutCracker generates programs well within this hardware capacity, the structural overhead consumes on-chip memory without triggering resource contention or impacting end-to-end performance.

Benefits of specialized lowering. Enabling lowering rules for specialized hardware backends yields significant performance gains. On the BlueField 2 platform, leveraging the fixed function accelerators (`+Lowering (CPU+FFA)`) provides substantial improvements in throughput, specifically 7.8× for Compression, 2.6× for File Verification, and 3.6× for the DNS Filter. As illustrated in Figure 13, the use of these fixed-function accelerators also reduces median latency across the board. Compression benefits the most from this approach, achieving a 6.3× speedup. This result is followed by a 1.7× speedup for regular expression matching in the DNS Filter, and a more modest 1.2× improvement for SHA256 hashing in the File Verification. Compared to BlueField-2, BlueField-3 provides fewer fixed-function accelerators but adds DPA cores as on-path processors. Despite exposing up to 190 hardware threads, the DPA subsystem suffers from higher per-packet latency and lower bandwidth. As shown in Figure 14 and Figure 13, mapping File Verification and the DNS Filter to the DPA results in severe throughput regressions, achieving only 0.2× and 0.6× of the baseline throughput, respectively. This architectural bottleneck also manifests as dramatically higher median latencies for these compute-heavy tasks, with execution times surging by 14× for the DNS Filter and over 60× for File Verification. These results justify the mapping policy of NutCracker: the compiler avoids placing latency- or bandwidth-intensive tasks on the DPA unless compiled under energy optimization mode.

Limits of compile-time performance estimation. The ablation study also reveals scenarios where the static compiler is not omnipotent. For IngressINT, throughput drops significantly starting from control flow flattening. Lacking runtime context, the compiler blindly lowers the single-entry match-action table into a full DPDK hash table, incurring massive iteration overhead over the manually optimized baseline. This behavior demonstrates that runtime could fundamentally alter the expected performance, highlighting the need to integrate dynamic factors such as table occupancy into the joint cost model. Interestingly, the DPA utilizes a simpler loop-based lookup, which inadvertently allows it to match the throughput of the ARM cores and minimize the expected latency gap.

7.5 Effectiveness of Compilation Modes

We use NutCracker 's `-Othroughput` and `-Oenergy` mode on four applications to demonstrate its ability to generate DSA-aware mappings. Table 3 summarizes the mappings generated for each application.

		`-Othroughput`	`-Oenergy`	
I-INT	Mapping	16 ARM cores	16 DPA cores	
	Thp (Gbps)	9.91	10.26	
	Energy	High	Medium	
NAT	Mapping	6 NIC flow tables		16 DPA cores
	Thp (Gbps)	74.4		9.3
	Energy	Low		Medium
SYN Flood	Mapping	12 NIC flow tables 3 NIC flow counters		16 DPA cores
	Thp (Gbps)	74.3		8.6
	Energy	Low		Medium
DDoS	Mapping	(Metering) 4 NIC flow tables (DNS Filter) 16 ARM cores		(Metering) 16 ARM cores (DNS Filter) 16 DPA cores
	Thp (Gbps)	(Metering) 37.4 (DNS Filter) 2.3		(Metering) 10.7 (DNS Filter) 1.3
	Energy	High		High

Table 3. Mapping variants, throughput, and energy consumption for applications compiled with different modes.

Energy-efficient DSA exploration with `-Oenergy`. The `-Oenergy` mode explicitly prioritizes mappings that utilize energy-efficient DSAs, such as the DPA or NIC ASICs. If there is more than one viable mapping that uses these DSAs, the compiler emits all candidate configurations for developers to select from.

Multi-path mapping. Applications with divergent datapath requirements necessitate sophisticated reasoning in mapping. For instance, the DDoS Detection in Table 3 performs TCP metering alongside DNS filtering. When compiled with `-Othroughput` mode, NutCracker deprioritizes DPA due to its limited bandwidth capacity. Instead, it produces a NIC-ARM mapping to maximize throughput. Conversely, under the `-Oenergy` mode, NutCracker emits both NIC-ARM and DPA-ARM configurations because both mappings inevitably require the highly power-consumptive general-purpose ARM cores.

8 Related Work

Compilers for in-network devices. Programmable switch compilers target fixed-stage match-action pipelines, addressing pipeline partitioning [19, 44], resource allocation under hardware constraints [15, 16, 49], and cross-device program distribution [9, 14, 57]. FPGA-based network compilers [18, 55] compile packet-processing logic into reconfigurable hardware, focusing on parser construction, table layout, and pipeline parallelism. While these tools target high-performance packet processing, these compilers assume either homogeneous match-action pipelines or uniform reconfigurable logic. In contrast, DPUs comprise heterogeneous DSAs, requiring compilers to coordinate execution across diverse architectural domains and programming models.

Performance modeling. Prior work models the performance of specialized network hardware to guide system design and optimization. Performance interfaces [27] expose hardware accelerator behavior, enabling developers to simulate workloads and make informed design decisions. Clara [39] uses machine learning to predict the performance of a network function when ported to a target DPU. LogNIC [17] analyzes the performance characteristics of a DPU-offloaded program at high-level, abstracting away low-level device details. Pipeleon [54] uses runtime profiling of P4 programs on DPUs to optimize program layout. NutCracker could benefit from these approaches to develop a more accurate and adaptive cost model.

Optimization for heterogeneous compilation. Recent work applies program synthesis techniques to optimize compilation for heterogeneous hardware targets. Lakeroad [46] applies sketch-guided synthesis [47] to map code onto configurable primitives like DSPs, balancing programmability and efficiency. Churchroad [46] extends Lakeroad by removing the sketch requirement and employing equality saturation [51] to systematically explore equivalent program rewrites. In programmable switches, synthesis methods like counterexample-guided inductive synthesis [1] and Satisfiability Modulo Theory (SMT) solving [5] are used in resource allocation [15, 16] and safe update planning [38] in P4 pipelines. NutCracker could integrate similar techniques to automatically derive rewrite rules, improving the completeness and generality of progressive lowering.

Programming abstractions for in-network devices. Prior works advocate programming languages [10, 24, 37, 48, 50] and IRs [28, 42] for in-network devices. Works like Coyote v2 [40] also introduce a new abstraction layer to reduce programming effort for devices that typically expose low-level programming interfaces. NutCracker is complementary to these works, introducing a new abstraction layer for domain-specific architectures to reduce the programming effort of using hybrid DPUs.

9 Conclusion

We present NutCracker, a compiler framework that enables portable and efficient programming for hybrid DPUs with heterogeneous DSAs. It combines a class-based vDSA abstraction, rule-based progressive lowering, and cost-guided mapping optimization to generate hardware-dependent implementations from high-level code. Evaluated on a range of DPU workloads, NutCracker reduces developer effort by up to 93%. For seven of the eight evaluated applications, it delivers performance within 3% of handcrafted baselines. Additionally, NutCracker outperforms Alkali's synthesis-based approach by up to 47.3×, while accelerating compilation by up to 31.9× to match the rapid turnaround expected of standard industrial toolchains.

Acknowledgments

We thank our shepherd Shaojie Xiang and the anonymous reviewers for their helpful feedback. This work was supported by the Singapore Ministry of Education, under the Academic Research Fund Tier 2 grant MOE-T2EP20222-0016.

References

[1] Alessandro Abate, Cristina David, Pascal Kesseli, Daniel Kroening, and Elizabeth Polgreen. 2018. Counterexample guided inductive synthesis modulo theories. In *International Conference on Computer Aided Verification*. Springer, 270–288.

[2] AMD. 2023. AMD PENSANDO 2ND GENERATION PLUS ("GIGLIO") DPU. https://www.amd.com/content/dam/amd/en/documents/pensando-technical-docs/product-briefs/pensando-giglio-product-brief.pdf.

[3] AMD. 2023. AMD PENSANDO 2ND GENERATION ("ELBA") DPU. https://www.amd.com/content/dam/amd/en/documents/pensando-technical-docs/product-briefs/pensando-elba-product-brief.pdf.

[4] Vojtech Aschenbrenner, John Shawger, and Sadman Sakib. 2024. FlexBSO: Flexible Block Storage Offload for Datacenters. arXiv:2409.02381 [cs.NI] https://arxiv.org/abs/2409.02381

[5] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2016. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`.

[6] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. 2014. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review* 44, 3 (2014), 87–95.

[7] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. *ACM SIGCOMM Computer Communication Review* 43, 4 (2013), 99–110.

[8] Zihao Chang, Jiaqi Zhu, Haifeng Sun, Yunlong Xie, Kan Shi, Ninghui Sun, Yungang Bao, and Sa Wang. 2025. Poby: SmartNIC-accelerated image provisioning for coldstart in clouds. In *Proceedings of USENIX ATC* (Boston, MA, USA) *(USENIX ATC '25)*. USENIX Association, USA, Article 2, 19 pages.

[9] Xiang Chen, Qingjiang Xiao, Hongyan Liu, Qun Huang, Dong Zhang, Xuan Liu, Longbing Hu, Haifeng Zhou, Chunming Wu, and Kui Ren. 2024. Eagle: Toward scalable and near-optimal network-wide sketch deployment in network measurement. In *Proceedings of the ACM SIGCOMM 2024 Conference*. 291–310.

[10] Sean Choi, Muhammad Shahbaz, Balaji Prabhakar, and Mendel Rosenblum. 2020. λ-nic: Interactive serverless compute on programmable smartnics. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 67–77.

[11] Sharad Chole, Andy Fingerhut, Sha Ma, Anirudh Sivaraman, Shay Vargaftik, Alon Berger, Gal Mendelson, Mohammad Alizadeh, Shang-Tse Chuang, Isaac Keslassy, et al. 2017. drmt: Disaggregated programmable switching. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. 1–14.

[12] Intel Corporation. 2025. Intel® Infrastructure Processing Unit (Intel® IPU) Adapter E2100. https://www.intel.com/content/www/us/en/products/details/network-io/ipu/adapter-e2100.html. Accessed: [Your access date].

[13] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. 2018. Azure accelerated networking:{SmartNICs} in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. 51–66.

[14] Jiaqi Gao, Ennan Zhai, Hongqiang Harry Liu, Rui Miao, Yu Zhou, Bingchuan Tian, Chen Sun, Dennis Cai, Ming Zhang, and Minlan Yu. 2020. Lyra: A cross-platform language and compiler for data plane programming on heterogeneous asics. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*. 435–450.

[15] Xiangyu Gao, Taegyun Kim, Michael D Wong, Divya Raghunathan, Aatish Kishan Varma, Pravein Govindan Kannan, Anirudh Sivaraman, Srinivas Narayana, and Aarti Gupta. 2020. Switch code generation using program synthesis. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*. 44–61.

[16] Xiangyu Gao, Divya Raghunathan, Ruijie Fang, Tao Wang, Xiaotong Zhu, Anirudh Sivaraman, Srinivas Narayana, and Aarti Gupta. 2023. Cat: A solver-aided compiler for packet-processing pipelines. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 72–88.

[17] Zerui Guo, Jiaxin Lin, Yuebin Bai, Daehyeok Kim, Michael Swift, Aditya Akella, and Ming Liu. 2023. Lognic: A high-level performance model for smartnics. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 916–929.

[18] Stephen Ibanez, Gordon Brebner, Nick McKeown, and Noa Zilberman. 2019. The p4-> netfpga workflow for line-rate packet processing. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 1–9.

[19] Lavanya Jose, Lisa Yan, George Varghese, and Nick McKeown. 2015. Compiling packet programs to reconfigurable switches. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. 103–115.

[20] Jongyul Kim, Insu Jang, Waleed Reda, Jaeseong Im, Marco Canini, Dejan Kostić, Youngjin Kwon, Simon Peter, and Emmett Witchel. 2021. Linefs: Efficient smartnic offload of a distributed file system with pipeline parallelism. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 756–771.

[21] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.* IEEE, 75–86.

[22] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2–14.

[23] Yanfang Le, Hyunseok Chang, Sarit Mukherjee, Limin Wang, Aditya Akella, Michael M Swift, and TV Lakshman. 2017. UNO: Uniflying host and smart NIC offload for flexible packet processing. In *Proceedings of the 2017 Symposium on Cloud Computing*. 506–519.

[24] Bojie Li, Kun Tan, Layong Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. 2016. Clicknp: Highly flexible and high performance network processing with reconfigurable hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference*. 1–14.

[25] Jiaxin Lin, Zhiyuan Guo, Mihir Shah, Tao Ji, Yiying Zhang, Daehyeok Kim, and Aditya Akella. 2025. Enabling Portable and {High-Performance}{SmartNIC} Programs with Alkali. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. 107–126.

[26] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. 2019. Offloading distributed applications onto smartnics using ipipe. In *Proceedings of the ACM Special Interest Group on Data Communication*. 318–333.

[27] Jiacheng Ma, Rishabh Iyer, Sahand Kashani, Mahyar Emami, Thomas Bourgeat, and George Candea. 2024. Performance interfaces for hardware accelerators. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 855–874.

[28] Kingshuk Majumder and Uday Bondhugula. 2023. HIR: An mlir-based intermediate representation for hardware accelerator description. In *Proceedings of the 28th ACM International Conference on Architectural*

Support for Programming Languages and Operating Systems, Volume 4. 189–201.

[29] Marvell. 2020. Marvell LiquidIO III SmartNIC Solutions. https://www.marvell.com/content/dam/marvell/en/public-collateral/embedded-processors/marvell-liquidio-III-solutions-brief.pdf.

[30] Marvell. 2020. Marvell OCTEON DPUs. https://www.marvell.com/products/data-processing-units.html.

[31] NVIDIA. 2023. NVIDIA BlueField-2 DPU. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-2-dpu.pdf.

[32] NVIDIA. 2023. NVIDIA BlueField-3 DPU. https://resources.nvidia.com/en-us-accelerated-networking-resource-library/datasheet-nvidia-bluefield.

[33] NVIDIA. 2024. NVIDIA Spectrum-X Ethernet Networking Platform Datasheet. https://resources.nvidia.com/en-us-networking-ai/networking-ethernet-1.

[34] NVIDIA. 2025. NVIDIA DPA Subsystem. https://docs.nvidia.com/doca/sdk/dpa+subsystem/index.html.

[35] P4 Language Consortium. 2023. P4MLIR: MLIR-based Intermediate Representation for P4. https://github.com/p4lang/p4mlir-incubator. Accessed: September 13, 2025.

[36] P4.org Architecture Working Group. 2023. Portable NIC Architecture (PNA) Specification. https://p4.org/p4-spec/docs/PNA.html. Accessed: 2025-09-12.

[37] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. 2018. Floem: A programming system for {NIC-Accelerated} network applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 663–679.

[38] Yiming Qiu, Ryan Beckett, and Ang Chen. 2023. Synthesizing runtime programmable switch updates. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 613–628.

[39] Yiming Qiu, Qiao Kang, Ming Liu, and Ang Chen. 2020. Clara: Performance clarity for smartnic offloading. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*. 16–22.

[40] Benjamin Ramhorst, Dario Korolija, Maximilian Jakob Heer, Jonas Dann, Luhao Liu, and Gustavo Alonso. 2025. Coyote v2: Raising the Level of Abstraction for Data Center FPGAs. *arXiv preprint arXiv:2504.21538* (2025).

[41] Henry N Schuh, Weihao Liang, Ming Liu, Jacob Nelson, and Arvind Krishnamurthy. 2021. Xenic: SmartNIC-accelerated distributed transactions. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 740–755.

[42] Amirali Sharifian, Reza Hojabr, Navid Rahimi, Sihao Liu, Apala Guha, Tony Nowatzki, and Arrvindh Shriraman. 2019. μir-an intermediate representation for transforming and optimizing the microarchitecture of application accelerators. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 940–953.

[43] Rajath Shashidhara, Tim Stamler, Antoine Kaufmann, and Simon Peter. 2022. {FlexTOE}: Flexible {TCP} offload with {Fine-Grained} parallelism. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 87–102.

[44] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. 2016. Packet transactions: High-level programming for line-rate switches. In *Proceedings of the 2016 ACM SIGCOMM Conference*. 15–28.

[45] Gus Henry Smith, Colin Knizek, Daniel Petrisko, Zachary Tatlock, Jonathan Balkind, Gilbert Louis Bernstein, Haobin Ni, and Chandrakana Nandi. 2024. Scaling Program Synthesis Based Technology Mapping with Equality Saturation. *arXiv preprint arXiv:2411.11036* (2024).

[46] Gus Henry Smith, Benjamin Kushigian, Vishal Canumalla, Andrew Cheung, Steven Lyubomirsky, Sorawee Porncharoenwase, René Just, Gilbert Louis Bernstein, and Zachary Tatlock. 2024. Fpga technology mapping using sketch-guided program synthesis. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 416–432.

[47] Armando Solar-Lezama. 2009. The sketching approach to program synthesis. In *Asian symposium on programming languages and systems*. Springer, 4–13.

[48] John Sonchack, Devon Loehr, Jennifer Rexford, and David Walker. 2021. Lucid: A language for control in the data plane. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. 731–747.

[49] Haifeng Sun, Qun Huang, Jinbo Sun, Wei Wang, Jiaheng Li, Fuliang Li, Yungang Bao, Xin Yao, and Gong Zhang. 2024. AutoSketch: Automatic Sketch-Oriented Compiler for Query-driven Network Telemetry. In *Proc. of USENIX NSDI*.

[50] Haifeng Sun, Bing Liu, Taixu Tian, Jinbo Sun, Jintao He, Qun Huang, Luyou He, Xuan Wang, Feng Gao, Liguo Wang, Xiangcan Xu, Junyi Guo, Xiaoping Zhu, and Yongqiang Yang. 2025. Rearchitecting Programmable Networks For In-Network Computing: From Hardware To Language. In *Proc. of ACM EuroSys*.

[51] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality saturation: a new approach to optimization. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 264–276.

[52] UTSN. 2025. Alkali GitHub Repository. https://github.com/utnslab/Alkali.

[53] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. Egg: Fast and extensible equality saturation. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–29.

[54] Jiarong Xing, Yiming Qiu, Kuo-Feng Hsu, Songyuan Sui, Khalid Manaa, Omer Shabtai, Yonatan Piasetzky, Matty Kadosh, Arvind Krishnamurthy, TS Eugene Ng, et al. 2023. Unleashing SmartNIC packet processing performance in P4. In *Proceedings of the ACM SIGCOMM 2023 Conference*. 1028–1042.

[55] Abbas Yazdinejad, Ali Bohlooli, and Kamal Jamshidi. 2018. P4 to sdnet: Automatic generation of an efficient protocol-independent packet parser on reconfigurable hardware. In *2018 8th International Conference on Computer and Knowledge Engineering (ICCKE)*. IEEE, 159–164.

[56] Abd-El-Aziz Zayed and Christophe Dubach. 2025. DialEgg: Dialect-Agnostic MLIR Optimizer using Equality Saturation with Egglog. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*. 271–283.

[57] Kaiyuan Zhang, Danyang Zhuo, and Arvind Krishnamurthy. 2020. Gallium: Automated software middlebox offloading to programmable switches. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*. 283–295.

[58] Yihong Zhang, Yisu Remy Wang, Oliver Flatt, David Cao, Philip Zucker, Eli Rosenthal, Zachary Tatlock, and Max Willsey. 2023. Better together: Unifying datalog and equality saturation. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 468–492.

[59] Jiechen Zhao, Ran Shu, Lei Qu, Ziyue Yang, Natalie Enright Jerger, Derek Chiou, Peng Cheng, and Yongqiang Xiong. 2024. Smartnic-enabled live migration for storage-optimized vms. In *Proceedings of the 15th ACM SIGOPS Asia-Pacific Workshop on Systems*. 45–52.

[60] Hao Zheng, Xin Yan, Wenbo Li, Jiaqi Zheng, Xiaoliang Wang, Qingqing Zhao, Luyou He, Xiaofei Lai, Feng Gao, Fuguang Huang, Wanchun Dou, Guihai Chen, and Chen Tian. 2025. When P4 Meets Run-to-completion Architecture. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. USENIX Association, Philadelphia, PA, 1487–1505. https://www.usenix.org/conference/nsdi25/presentation/zheng-hao

Appendix A End-to-End Compilation Walkthrough

To demonstrate NutCracker 's end-to-end compilation pipeline, this appendix traces the compilation phases of a port-based flow counter. This application filters TCP flows by destination port and tracks their packet counts using a stateful counter indexed by packing the flow's 5-tuple hash with the destination port. The following subsections trace this application's compilation from its high-level P4 implementation, lowered by NutCracker, down to a declarative pipeline on the BlueField-3 DPU.

A.1 Frontend Interfaces and Hardware Abstraction

To accommodate diverse developer backgrounds and integrate with existing network stacks, NutCracker supports both C and P4-16 as frontend languages. The framework abstracts hardware-specific accelerator details through standardized library headers (`nc.h` for C and `nc.p4` for P4). These headers define `extern` interfaces for complex accelerators, such as stateful Counter arrays and hardware hashing engines (e.g., `Hash5Tuple`).

Defining accelerators as `extern` functions decouples functionality from DPU architecture, providing flexibility for subsequent compiler reasoning. Since DPU designs are highly vendor-specific, a hardware accelerator available on one target may be absent on another. For instance, the Regular Expression Engine exists on NVIDIA BlueField-2 and Marvell OCTEON TX2, yet is absent on NVIDIA BlueField-3. By establishing this opaque functional boundary via `extern`, the frontend completely circumvents the need for hardware-specific capability probing. This design guarantees both functional correctness and performance portability: During target-specific lowering, NutCracker resolves each invocation by mapping it directly to a native hardware accelerator if present, or by seamlessly linking a software fallback executed on the DPU's general-purpose cores.

Listing 1 illustrates a representative P4 implementation. Within the `hash_and_count` action, the application computes a 5-tuple hash. The program then extracts the lower 16 bits of the hash, performs a logical left shift, and applies a bitwise `OR` with the TCP destination port to pack the values into a single 32-bit counter index. While this logic is easily expressed in high-level software, arbitrary bitwise arithmetic is often fundamentally incompatible with the Instruction Set Architectures (ISAs) of target DSAs. For instance, the flow action exposed by the DOCA Flow API supports only a strict subset of arithmetic operations. This ISA incompatibility makes compilation extremely challenging.

```
// --- nc.p4 ---
extern Counter<I> {
    Counter(bit<32> size);
    void count(in I index);
}
extern Hash5Tuple {
    Hash5Tuple();
    void apply(out bit<32> result, in bit<32> srcA, in bit<32> dstA,
               in bit<16> srcP, in bit<16> dstP, in bit<8> proto);
}
// --- Application Code ---
#include <nc.p4>
struct metadata {
  bit<32> hash_index;
}
control Ingress(inout headers h, inout meta m) {
  Hash5Tuple() hash;
  Counter<bit<32>>(1024) tcp_count;
  action hash_and_count() {
    hash.apply(
      m.hash_index, h.ipv4.srcA, h.ipv4.dstA,
      h.tcp.srcP, h.tcp.dstP, h.ipv4.prot
    );
    bit<32> lower_hash = meta.hash_index & 32w0x0000FFFF;
    bit<32> shifted_hash = lower_hash << 8;
    meta.hash_index = shifted_hash | (bit<32>)hdr.tcp.dst_port;
    tcp_count.count(packed_index);
  }
  table tcp_hash_tbl {
    key = { h.tcp.dstP: exact; }
    actions = { hash_and_count; }
  }
  apply {
    if (h.ipv4.isValid() && h.tcp.isValid()) {
      tcp_hash_tbl.apply();
    }
  }
}
NC_PIPELINE(Parser(), Ingress(), Deparser()) main;
```

Listing 1. P4 program for port-based flow counting.

A.2 Coarse-grained Lowering to vDSA

NutCracker first rewrites precompiled frontend representations (P4HIR or LLVM IR) into core vDSAs. To accommodate diverse execution models across target vDSAs, such as table-driven pipelines or run-to-completion cores, the compiler first flattens nested and conditional control flows into a strict control flow graph (CFG). Each node within this CFG represents a basic block: a linear, branch-free sequence of operations maintained in Static Single Assignment (SSA) form. Through a dedicated MLIR transformation pass, NutCracker maps the frontend instructions within each basic block down to available core vDSA classes.

Listing 2 illustrates the result of this coarse-grained lowering specifically for the core `vDRMT` dialect for the virtual Disaggregated Reconfigurable Match-action Table. The original P4 program is decomposed into explicitly chained basic blocks, modeled here as match-action tables (`@blk1` through `@blk5`). To unambiguously define the execution path and state transitions between pipeline stages, every block enforces an explicit terminator (e.g., `vdrmt.next @blk2` to advance to the subsequent table, or `vdrmt.next @next_port` to be flushed onto the wire via NIC port).

```
vdrmt.extern @Counter {
  vdrmt.func @Counter(i32, !CounterType)
  vdrmt.func @count(i32)
}
vdrmt.extern @Hash5Tuple {
  vdrmt.func @Hash5Tuple()
```

```
  vdrmt.func @apply(!vdrmt.ref<i32>, i32, i32, i16, i16, i8)
}
vdrmt.control @Ingress(%arg0, %arg1)() {
  %hash = vdrmt.instantiate @Hash5Tuple()
  %c1024 = vdrmt.const 1024
  %syn_count = vdrmt.instantiate @Counter(%c1024)
  vdrmt.table @blk1 {
    vdrmt.table_key = {
      %ipv4_field_ref = vdrmt.struct_extract_ref %arg0["ipv4"]
      %ipv4_valid_ref = vdrmt.struct_extract_ref %ipv4_field_ref["__valid"]
      %val = vdrmt.read %ipv4_valid_ref
      %valid = vdrmt.const #valid
      %eq = vdrmt.cmp eq, %val, %valid
      vdrmt.match_key #exact, %eq
      %tcp_field_ref = vdrmt.struct_extract_ref %arg0["tcp"]
      %tcp_valid_ref = vdrmt.struct_extract_ref %tcp_field_ref["__valid"]
      %val_1 = vdrmt.read %tcp_valid_ref
      %valid_2 = vdrmt.const #valid
      %eq_3 = vdrmt.cmp eq, %val_1, %valid_2
      vdrmt.match_key #exact, %eq_3
    }
    vdrmt.table_actions {
      vdrmt.table_action @next_pipe() { vdrmt.next @blk2 }
      vdrmt.table_action @next_port() { vdrmt.next @next_port }
    }
    vdrmt.table_default_action { vdrmt.next @next_port }
  }
  vdrmt.table @blk2 {
    vdrmt.table_key = {
      %val = vdrmt.read %arg0
      %tcp = vdrmt.struct_extract %val["tcp"]
      %dst_port = vdrmt.struct_extract %tcp["dst_port"]
      vdrmt.match_key #exact %dst_port
    }
    vdrmt.table_actions {
      vdrmt.table_action @next_pipe() { vdrmt.next @blk3 }
    }
    vdrmt.table_default_action { vdrmt.next @next_port }
  }
  vdrmt.table @blk3 {
    vdrmt.table_default_action {
      %hash_index_field_ref = vdrmt.struct_extract_ref %arg1["hash_index"]
      %result_out_arg = vdrmt.variable ["result_out_arg"]
      %val = vdrmt.read %arg0
      %ipv4 = vdrmt.struct_extract %val["ipv4"]
      %src_addr = vdrmt.struct_extract %ipv4["src_addr"]
      %val_0 = vdrmt.read %arg0
      %ipv4_1 = vdrmt.struct_extract %val_0["ipv4"]
      %dst_addr = vdrmt.struct_extract %ipv4_1["dst_addr"]
      %val_2 = vdrmt.read %arg0
      %tcp = vdrmt.struct_extract %val_2["tcp"]
      %src_port = vdrmt.struct_extract %tcp["src_port"]
      %val_3 = vdrmt.read %arg0
      %tcp_4 = vdrmt.struct_extract %val_3["tcp"]
      %dst_port = vdrmt.struct_extract %tcp_4["dst_port"]
      %val_5 = vdrmt.read %arg0
      %ipv4_6 = vdrmt.struct_extract %val_5["ipv4"]
      %protocol = vdrmt.struct_extract %ipv4_6["protocol"]
      vdrmt.vffa.call @5TupleHash::@apply(%hash, %res_ref, %src_addr, %dst_addr, %src_port, %dst_port, %proto)
      vdrmt.next @blk4
    }
  }
  vdrmt.table @blk4 {
    vdrmt.table_default_action {
      %val = vdrmt.read %arg1
      %hash_index = vdrmt.struct_extract %val["hash_index"]
      %c65535_b32i = vdrmt.const 65535
      %and = vdrmt.binop(and, %hash_index, %c65535_b32i)
      %lower_hash = vdrmt.variable ["lower_hash", init]
      vdrmt.assign %and, %lower_hash
      %c8 = vdrmt.const 8
      %val_0 = vdrmt.read %lower_hash
      %shl = vdrmt.shl(%val_0, %c8)
      %shifted_hash = vdrmt.variable ["shifted_hash", init]
      vdrmt.assign %shl, %shifted_hash
      %hash_index_field_ref = vdrmt.struct_extract_ref %arg1["hash_index"]
      %val_1 = vdrmt.read %arg0
      %tcp = vdrmt.struct_extract %val_1["tcp"]
      %dst_port = vdrmt.struct_extract %tcp["dst_port"]
      %cast_2 = vdrmt.cast(%dst_port)
      %val_3 = vdrmt.read %shifted_hash
      %or = vdrmt.binop(or, %val_3, %cast_2)
      vdrmt.assign %or, %hash_index_field_ref
      vdrmt.next @blk5
    }
  }
  vdrmt.table @blk5 {
    vdrmt.table_default_action {
      %val_4 = vdrmt.read %arg1
      %hash_index_5 = vdrmt.struct_extract %val_4["hash_index"]
      vdrmt.vffa.call @Counter::@count (%tcp_count, %hash_index_5)
      vdrmt.next @next_port
    }
  }
}
```

Listing 2. vDRMT intermediate representation of the port-based flow counter after coarse-grained lowering.

A.3 Fine-grained Lowering with DialEgg

NutCracker then rewrites intermediate representations in core vDSA dialects into target-specific vDSA instances that reflect the physical constraints on a specific DPU. To achieve this, the compiler employs DialEgg, a novel framework that integrates Equality Saturation into an MLIR pass to enable seamless cross-dialect rewriting. It maximizes the probability of successfully lowering idealized vDSA operations into the specific, restrictive primitives natively supported by the target DSA. These idealized operations may assume unconstrained matching, memory access, or arithmetic capabilities that are not present on the physical DSA.

While the EqSat engine provides a highly robust framework for syntax-driven structural rewriting, the semantic equivalence between MLIR operations must be explicitly defined by human developers. The engine operates purely on syntactic pattern matching and does not natively synthesize or verify semantic equivalence. For instance, as demonstrated

in Listing 3, developers must manually encode the mathematical insight that a logical left shift by 8 bits (`vdrmt_shl`) is functionally equivalent to a hardware crossbar concatenation with an 8-bit zero-padded constant (`bf3drmt_concat`). Once these rules are defined, the engine applies them by treating the e-graph as a relational database. It identifies matches by executing a conjunctive query (similar to a Datalog rule) over the e-nodes. When a match is found, the engine performs a union operation to represent the equality between the existing expression and the hardware-native primitive. Automating the inference of these hardware-specific rewrite rules is a promising future direction for NutCracker.

```
;; Rule 1: AND with 0xFFFF -> Slice
(rule
  ((= ?and_op (vdrmt_binop ?hash_val ?mask_const (
       NamedAttr "kind" (IntegerAttr 10 (I32))) (i32)))
   (= ?mask_const (vdrmt_constant (NamedAttr "value" (
       IntegerAttr 65535 (I32))) (i32))))
  ((union ?and_op
      (bf3drmt_slice ?hash_val 15 0 (i16))))
)
;; Rule 2: SHL by 8 -> Concat with Zero
(rule
  ((= ?shl_op (vdrmt_shl ?operand ?shift (i32)))
   (= ?shift (vdrmt_constant (NamedAttr "value" (
       IntegerAttr 8 (I32))) (i32)))
   (= ?operand (bf3drmt_slice ?x ?hi ?lo (i16))))
  ((union ?shl_op
          (bf3drmt_concat ?operand (vdrmt_constant (
               NamedAttr "value" (IntegerAttr 0 (I16)))
                (i16)) (i32))))
)
;; Rule 3: OR with Zero-Padded Concat -> Concat
(rule
  ((= ?or_op (vdrmt_binop ?left ?right (NamedAttr "
       kind" (IntegerAttr 8 (I32))) (i32)))
   (= ?left (bf3drmt_concat ?upper ?zero_const (i32)))
   (= ?zero_const (vdrmt_constant (NamedAttr "value" (
       IntegerAttr 0 (I16))) (i16))))
  ((union ?or_op
          (bf3drmt_concat ?upper ?right (i32))))
)
```

Listing 3. Egglog rules to rewrite bitwise arithmetic in vDRMT dialect to BlueField-3 dRMT dialect.

After the EqSat engine applies these human-defined rulesets in the fine-grained lowering phase, NutCracker canonicalizes the program into the dialect of the concrete vDSA instance. As shown in Listing 4, the efficacy of fine-grained lowering is clearly evident within `@blk4`. Although the Network ASICs on the BlueField-3 lack native support for arbitrary bitwise arithmetic, NutCracker successfully eliminates these unsupported operations. By replacing the absent bitwise arithmetic with supported `bf3drmt.slice` and `bf3drmt.concat`, the compiler ensures that the packing logic is fully executable on the physical DSA.

```
bf3drmt.control @Ingress(%arg0, %arg1)() {
  bf3drmt.table @blk1 {
    bf3drmt.table_key = {
      %ipv4_field_ref = bf3drmt.struct_extract_ref %
          arg0["ipv4"]
      %ipv4_valid_ref = bf3drmt.struct_extract_ref %
          ipv4_field_ref["__valid"]
      %val = bf3drmt.read %ipv4_valid_ref
      %valid = bf3drmt.const #valid
      %eq = bf3drmt.cmp eq, %val, %valid
      bf3drmt.match_key #exact, %eq
      %tcp_field_ref = bf3drmt.struct_extract_ref %
          arg0["tcp"]
      %tcp_valid_ref = bf3drmt.struct_extract_ref %
          tcp_field_ref["__valid"]
      %val_1 = bf3drmt.read %tcp_valid_ref
      %valid_2 = bf3drmt.const #valid
      %eq_3 = bf3drmt.cmp eq, %val_1, %valid_2
      bf3drmt.match_key #exact, %eq_3
    }
    bf3drmt.table_actions {
      bf3drmt.table_action @next_pipe() { bf3drmt.next
          @blk2 }
      bf3drmt.table_action @next_port() { bf3drmt.next
          @next_port }
    }
    bf3drmt.table_default_action { bf3drmt.next @
        next_port }
  }
  bf3drmt.table @blk2 {
    bf3drmt.table_key = {
      %tcp_field_ref = bf3drmt.struct_extract_ref %
          arg0["tcp"]
      %dst_port_ref = bf3drmt.struct_extract_ref %
          tcp_field_ref["dst_port"]
      %val = bf3drmt.read %dst_port_ref
      bf3drmt.match_key #exact %val
    }
    bf3drmt.table_actions {
      bf3drmt.table_action @next_pipe() { bf3drmt.next
          @blk3 }
    }
    bf3drmt.table_default_action { bf3drmt.next @
        next_port }
  }
  bf3drmt.table @blk3 type("hash") {
    bf3drmt.table_key = {
      %ipv4_ref_1 = bf3drmt.struct_extract_ref %arg0["
          ipv4"]
      %src_addr_ref = bf3drmt.struct_extract_ref %
          ipv4_ref_1["src_addr"]
      %src_addr_val = bf3drmt.read %src_addr_ref
      bf3drmt.match_key #exact, %src_addr_val
      %ipv4_ref_2 = bf3drmt.struct_extract_ref %arg0["
          ipv4"]
      %dst_addr_ref = bf3drmt.struct_extract_ref %
          ipv4_ref_2["dst_addr"]
      %dst_addr_val = bf3drmt.read %dst_addr_ref
      bf3drmt.match_key #exact, %dst_addr_val
      %tcp_ref_1 = bf3drmt.struct_extract_ref %arg0["
          tcp"]
      %src_port_ref = bf3drmt.struct_extract_ref %
          tcp_ref_1["src_port"]
      %src_port_val = bf3drmt.read %src_port_ref
      bf3drmt.match_key #exact, %src_port_val
      %tcp_ref_2 = bf3drmt.struct_extract_ref %arg0["
          tcp"]
      %dst_port_ref = bf3drmt.struct_extract_ref %
          tcp_ref_2["dst_port"]
      %dst_port_val = bf3drmt.read %dst_port_ref
      bf3drmt.match_key #exact, %dst_port_val
      %ipv4_ref_3 = bf3drmt.struct_extract_ref %arg0["
          ipv4"]
      %proto_ref = bf3drmt.struct_extract_ref %
          ipv4_ref_3["protocol"]
      %proto_val = bf3drmt.read %proto_ref
      bf3drmt.match_key #exact, %proto_val
    }
    bf3drmt.table_default_action {
      %c0 = bf3drmt.const 0
      bf3drmt.set_metadata %meta.pkt_meta, %c0
      bf3drmt.next @blk4
    }
    bf3drmt.table_action @apply(%idx) {
      bf3drmt.set_metadata %meta.pkt_meta, %idx
      bf3drmt.next @blk4
    }
    bf3drmt.entries = {0..NB_ENTRIES : @apply() }
  }
```

```
69    bf3drmt.table @blk4 {
70      bf3drmt.table_default_action {
71        %hash_ref = bf3drmt.struct_extract_ref %arg1["
              hash_index"]
72        %hash_val = bf3drmt.read %hash_ref
73        %tcp_ref = bf3drmt.struct_extract_ref %arg0["tcp
              "]
74        %dst_port_ref = bf3drmt.struct_extract_ref %
              tcp_ref["dst_port"]
75        %dst_port_val = bf3drmt.read %dst_port_ref
76        %lower_hash = bf3drmt.slice %hash_val [15:0]
77        %packed_32 = bf3drmt.concat %lower_hash, %
              dst_port_val
78        bf3drmt.assign %hash_ref, %packed_32
79        bf3drmt.next @blk5
80      }
81    }
82    bf3drmt.table @blk5 {
83      bf3drmt.match_key = {
84        %hash_ref = bf3drmt.struct_extract_ref %arg1["
              hash_index"]
85        %hash_val = bf3drmt.read %hash_ref
86        bf3drmt.match_key #exact, %hash_val
87      }
88      bf3drmt.monitor = #bf3drmt.monitor_type<NON_SHARED
            >
89      bf3drmt.table_action @count(%idx) {
90        bf3drmt.next @next_port
91      }
92      bf3drmt.entries = { 0..NB_ENTRIES : @count() }
93    }
94  }
```

Listing 4. The canonicalized `bf3drmt` MLIR with eliminated arithmetic operations.

A.4 Backend Code Generation

The final stage of NutCracker generates source code from the concrete vDSA dialect for further compilation via vendor-specific toolchains or GCC. While the generation process varies by target, the compiler generally populates a vDSA-specific template to program the underlying hardware. For dialects like vRMT and vDPP, this phase can emit P4 code or LLVM IR, respectively.

We use the code generation for the Network ASICs on BlueField-3 as an example. As shown in Listing 5, the compiler translates high-level `bf3drmt` operations into the appropriate DOCA Flow structures. First, the backend maps field matching to `doca_flow_match` structures. This step specifies which header fields to match on and the match type, such as exact or ternary. Next, it translates standard action logic into `doca_flow_actions` to specify field modifications, such as rewriting a destination IP address. For more complex bitwise operations, the compiler converts `bf3drmt.slice` and `bf3drmt.concat` into bitwise copy actions within `doca_flow_action_descs`. These descriptors allow the hardware crossbar to extract and pack arbitrary bit ranges from headers into metadata. Finally, NutCracker maps stateful operations to `doca_flow_monitor` handles to manage hardware counters.

The resulting source code includes the control plane handlers to instantiate, initialize, and manage entries for each hardware table. This generated C code is compiled via GCC and linked against `libDOCA`. At execution time, the NutCracker runtime uses these handlers to install entries and manage state directly on the Network ASICs.

```
1  doca_error_t create_blk1_pipe(struct doca_flow_port *
       port,
2      int port_id,
3      struct doca_flow_pipe *next_pipe,
4      struct doca_flow_pipe **pipe)
5  {
6    struct doca_flow_match match = {0};
7    struct doca_flow_actions actions = {0}, *actions_arr
         [1];
8    struct doca_flow_fwd fwd = {0};
9    struct doca_flow_pipe_cfg *pipe_cfg;
10
11   match.outer.l3_type = DOCA_FLOW_L3_TYPE_IP4;
12   match.outer.ip4.next_proto = DOCA_FLOW_PROTO_TCP;
13
14   actions_arr[0] = &actions;
15
16   fwd.type = DOCA_FLOW_FWD_PIPE;
17   fwd.next_pipe = next_pipe;
18
19   doca_flow_pipe_cfg_create(&pipe_cfg, port);
20   set_flow_pipe_cfg(pipe_cfg, "blk1",
         DOCA_FLOW_PIPE_BASIC, true);
21   doca_flow_pipe_cfg_set_match(pipe_cfg, &match, NULL)
         ;
22   doca_flow_pipe_cfg_set_enable_strict_matching(
         pipe_cfg, true);
23   doca_flow_pipe_cfg_set_actions(pipe_cfg, actions_arr
         , NULL, NULL, 1);
24   doca_flow_pipe_create(pipe_cfg, &fwd, NULL, pipe);
25   return DOCA_SUCCESS;
26 }
27 doca_error_t create_blk2_pipe(struct doca_flow_port *
       port,
28     int port_id,
29     struct doca_flow_pipe *next_pipe,
30     struct doca_flow_pipe **pipe)
31 {
32   struct doca_flow_match match = {0};
33   struct doca_flow_actions actions = {0}, *actions_arr
         [1];
34   struct doca_flow_fwd fwd = {0};
35   struct doca_flow_pipe_cfg *pipe_cfg;
36
37   memset(&match, 0, sizeof(match));
38   memset(&actions, 0, sizeof(actions));
39   memset(&fwd, 0, sizeof(fwd));
40   memset(&status, 0, sizeof(status));
41
42   match.outer.tcp.l4_port.dst_port = 0xffff;
43
44   actions_arr[0] = &actions;
45
46   fwd.type = DOCA_FLOW_FWD_PIPE;
47   fwd.next_pipe = next_pipe;
48
49   doca_flow_pipe_cfg_create(&pipe_cfg, port);
50   set_flow_pipe_cfg(pipe_cfg, "blk2",
         DOCA_FLOW_PIPE_BASIC, false);
51   doca_flow_pipe_cfg_set_match(pipe_cfg, &match, NULL)
         ;
52   doca_flow_pipe_cfg_set_actions(pipe_cfg, actions_arr
         , NULL, NULL, 1);
53   doca_flow_pipe_create(pipe_cfg, &fwd, NULL, pipe);
54   return DOCA_SUCCESS;
55 }
56 doca_error_t create_blk3_pipe(struct doca_flow_port *
       port,
57     int port_id,
58     struct doca_flow_pipe *next_pipe,
59     struct doca_flow_pipe **pipe)
60 {
61   struct doca_flow_match match_mask = {0};
62   struct doca_flow_actions actions = {0}, *actions_arr
         [1];
```

```
63   struct doca_flow_fwd fwd = {0};
64   struct doca_flow_pipe_cfg *pipe_cfg;
65
66   match_mask.outer.l3_type = DOCA_FLOW_L3_TYPE_IP4;
67   match_mask.outer.ip4.src_ip = UINT32_MAX;
68   match_mask.outer.ip4.dst_ip = UINT32_MAX;
69   match_mask.outer.l4_type_ext =
         DOCA_FLOW_L4_TYPE_EXT_TCP;
70   match_mask.outer.tcp.l4_port.src_port = UINT32_MAX;
71   match_mask.outer.tcp.l4_port.dst_port = UINT32_MAX;
72
73   actions_arr[0] = &actions;
74
75   fwd.type = DOCA_FLOW_FWD_PIPE;
76   fwd.next_pipe = next_pipe;
77
78   doca_flow_pipe_cfg_create(&pipe_cfg, port);
79   set_flow_pipe_cfg(pipe_cfg, "blk3",
         DOCA_FLOW_PIPE_HASH, false);
80   doca_flow_pipe_cfg_set_nr_entries(pipe_cfg,
         NB_ENTRIES);
81   doca_flow_pipe_cfg_set_match(pipe_cfg, NULL, &
         match_mask);
82   doca_flow_pipe_cfg_set_actions(pipe_cfg, actions_arr
         , NULL, NULL, 1);
83   doca_flow_pipe_create(pipe_cfg, &fwd, NULL, pipe);
84   return DOCA_SUCCESS;
85 }
86 doca_error_t create_blk4_pipe(struct doca_flow_port *
       port,
87     int port_id,
88     struct doca_flow_pipe *next_pipe,
89     struct doca_flow_pipe **pipe)
90 {
91   struct doca_flow_match match = {0};
92   struct doca_flow_actions actions = {0}, *actions_arr
         [1];
93   struct doca_flow_action_descs descs = {0}, *
         descs_arr[2];
94   struct doca_flow_action_desc desc_arr[2] = {0};
95   struct doca_flow_fwd fwd = {0};
96   struct doca_flow_pipe_cfg *pipe_cfg;
97
98   desc_arr[0].type = DOCA_FLOW_ACTION_COPY;
99   desc_arr[0].field_op.src.field_string = "meta.data";
100  desc_arr[0].field_op.src.bit_offset =
         META_U32_BIT_OFFSET(0);
101  desc_arr[0].field_op.dst.field_string = "meta.data";
102  desc_arr[0].field_op.dst.bit_offset =
         META_U32_BIT_OFFSET(1) + 16;
103  desc_arr[0].field_op.width = 16;
104
105  desc_arr[1].type = DOCA_FLOW_ACTION_COPY;
106  desc_arr[1].field_op.src.field_string = "outer.tcp.
         dst_port";
107  desc_arr[1].field_op.src.bit_offset = 0;
108  desc_arr[1].field_op.dst.field_string = "meta.data";
109  desc_arr[1].field_op.dst.bit_offset =
         META_U32_BIT_OFFSET(1);
110  desc_arr[1].field_op.width = 16;
111
112  actions_arr[0] = &actions;
113  descs_arr[0] = &descs;
114  descs.nb_action_desc = 2;
115  descs.desc_array = desc_arr;
116
117  fwd.type = DOCA_FLOW_FWD_PIPE;
118  fwd.next_pipe = next_pipe;
119
120  doca_flow_pipe_cfg_create(&pipe_cfg, port);
121  set_flow_pipe_cfg(pipe_cfg, "blk4",
         DOCA_FLOW_PIPE_BASIC, false);
122  doca_flow_pipe_cfg_set_match(pipe_cfg, &match, NULL)
         ;
123  doca_flow_pipe_cfg_set_actions(pipe_cfg, actions_arr
         , NULL, descs_arr, 1);
124  doca_flow_pipe_create(pipe_cfg, &fwd, NULL, pipe);
125  return DOCA_SUCCESS;
126 }
127 doca_error_t create_blk5_pipe(struct doca_flow_port *
       port,
128     int port_id,
129     struct doca_flow_pipe *next_pipe,
130     struct doca_flow_pipe **pipe)
131 {
132   struct doca_flow_match match = {0};
133   struct doca_flow_actions actions = {0}, *actions_arr
         [1];
134   struct doca_flow_monitor monitor = {0};
135   struct doca_flow_fwd fwd = {0};
136   struct doca_flow_pipe_cfg *pipe_cfg;
137
138   match.meta.u32[0] = UINT32_MAX;
139
140   monitor.counter_type =
         DOCA_FLOW_RESOURCE_TYPE_NON_SHARED;
141
142   actions_arr[0] = &actions;
143
144   doca_flow_pipe_cfg_create(&pipe_cfg, port);
145   set_flow_pipe_cfg(pipe_cfg, "blk5",
         DOCA_FLOW_PIPE_HASH, false);
146   doca_flow_pipe_cfg_set_match(pipe_cfg, &match, NULL)
         ;
147   doca_flow_pipe_cfg_set_actions(pipe_cfg, actions_arr
         , NULL, NULL, 1);
148   doca_flow_pipe_cfg_set_monitor(pipe_cfg, &monitor);
149   doca_flow_pipe_cfg_set_nr_entries(pipe_cfg,
         NB_ENTRIES);
150   doca_flow_pipe_create(pipe_cfg, &fwd, NULL, pipe);
151   return DOCA_SUCCESS;
152 }
```

Listing 5. Generated DOCA Flow C code for table instantiation.