Precedence: Enabling Compact Program Layout By Table Dependency Resolution

Christopher Leet[1], Shenshen Chen[12], Kai Gao[3], Yang Richard Yang[12]

[1] Yale University, [2] Tongji University, [3] Sichuan University

ABSTRACT

The rise of the programmable switching ASIC has allowed switches to handle the complexity and diversity of modern networking programs while meeting the performance demands of modern networks. Exploitation of the flexibility of these switches, however, has exploded routing program size: recently proposed programs contain more than 100 [11] or even 1000 [10] tables. Realizing these programs in a programmable switch requires finding layouts with minimal depth: if a layout has more match-action stages than a switch's pipeline provides, the switch must recirculate, cutting throughput. Even if a layout fits a switch's pipeline, since most commercial pipelines cannot allocate memory freely to stages, non-compact pipelines can result in underloaded stages and significant memory underutilization. While inter-table control and data dependencies critically limit the ability of compilers to lay out tables compactly, no switch architecture which can fully resolve dependencies has been proposed. To address this problem, we introduce **precedence**, an extension of the RMT switching ASIC, which enables tables linked by dependencies to be executed in parallel or even out-of-order. Precedence can resolve nearly **70%** of switch.p4 [11]'s dependencies (a real-world routing program), reduce its pipeline depth by **48%**, and only modestly increases silicon area.

CCS CONCEPTS

• **Networks → Routers**; • **Hardware**;

1 INTRODUCTION

The rise of the programmable switching ASIC was a landmark development in modern networking. Programmability allows switches to handle the complexity and diversity of modern routing applications while achieving the performance required of modern networks. As programmers have increasingly exploited the flexibility promised by programmable switches, however, routing program size and complexity has ballooned. Recent routing programs like DC.p4 [23] (43 tables), switch.p4 [11] (163 tables) and HyPer4 [10] (1091 tables) have incredible sophistication.

Realizing a modern program on a programmable switch pipeline, however, requires finding a pipeline layout with minimal depth. Current pipeline targets, such as RMT [3], rarely have more than 32 match-action stages; a program layout which exceeds its switch target's depth forces the switch to recirculate packets, dropping throughput. Even when a program layout can fit into its target pipeline, there are good reasons for minimizing its depth. Longer pipelines have higher latency and must power more stages, resulting in high power consumption. Moreover, since most pipeline targets either disallow or strictly constrain stages from sharing memory, an unnecessarily long pipeline layout can lead to underutilization of each stage's resources, increasing its effective demand for costly SRAM and TCAM.

A critical limitation on the ability of compilers to generate compact datapath layouts is the inter-table data and control hazards, or **dependencies**, in routing programs. Under current pipeline architectures, if a table, T_1, has an output read by a second table, T_2, T_2 must be placed after T_1 in the datapath layout. Modern routing programs, reliant on complex systems of small, interconnected tables to perform computation, are burdened with cumbersome webs of dependencies. switch.p4 has 244 dependencies connecting its 163 tables!

To achieve compact pipeline layout, a pipeline architecture must allow parallel and even out of order execution of tables linked by dependencies, eliminating all constraints on pipeline layout except hardware constraints. Dependency resolution, however, raises the following challenges:

(1) **Correct parallel table execution.** Parallel execution of tables linked by dependencies can introduce race conditions [21] which must be handled correctly.
(2) **Efficient parallel table execution.** Switching ASICs must meet demanding throughput, latency, and power requirements. Any ASIC which permits parallelism must not compromise these requirements.

No public switching ASIC architecture resolves routing program dependencies. While work on speculative execution

SOSR '19, April 3–4, 2019, San Jose, CA, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6710-3/19/04…$15.00
https://doi.org/10.1145/3314148.3314348

in CPU/GPU based systems [8, 15, 17] has yielded insights into handling race conditions in parallelism, these insights cannot be directly applied to switching ASICs.

In this paper, we introduce **precedence**, an extension of the RMT architecture which **novelly** allows tables linked by both data and control dependencies to be executed in parallel and even out-of-order by speculative execution. Precedence's underlying idea is not new [20], but we are the first to introduce it to the design of switching ASICs. Precendence can resove nearly **70%** of switch.p4's dependencies, reduces DC.p4 and switch.p4's pipeline depths by **40%** and **48%**, and only requires a modest increase in silicon area.

2 PROBLEM STATEMENT

To allow better parallelism of logical tables, our objective is to resolve as many dependencies between logical programming tables as possible. We focus on extending the RMT [4] pipeline to resolve dependencies between logical tables in P4 [2] programs, but we emphasize that our approach is generic and could be applied to other logical pipeline descriptions and hardware layouts.

2.1 Pipeline Dependency Model

We model the graph of dependencies between logical routing tables with Jose et al.'s [13] notion of a Table Dependency Graph (TDG). A TDG is a directed, acyclic graph (DAG) representing a pipeline's logical tables as vertices and the dependency between them as directed edges.

The edges in a TDG can be grouped into four classes, corresponding to the four types of data and control dependency. [13, 20] Each class has different characteristics and requires a different resolution. These classes are:

- *Read-After-Write (RAW) data dependency:* Table 1 writes a field read by Table 2.
- *Write-After-Write (WAW) data dependency:* Table 1 writes a field subsequently written by Table 2.
- *Write-After-Read (WAR) data dependency:* Table 1 reads a field subsequently written by Table 2.
- *Control dependency:* Table 1 determines whether program control passes to Table 2 or not.

As an example, Figure 1 gives a TDG for the L2 L3 Simple MTag benchmark program implemented by Jose et al. Each table is depicted as a box, each control dependency as dashed edge, each RAW data dependencies as a red solid edge and the lone WAW dependency as a blue solid edge. Since there may be multiple types of dependencies between two tables, a TDG may be a multigraph.

2.2 Switch Architecture

We present precedence as an extension of the RMT switch architecture: - a real, high-performance, programmable switch

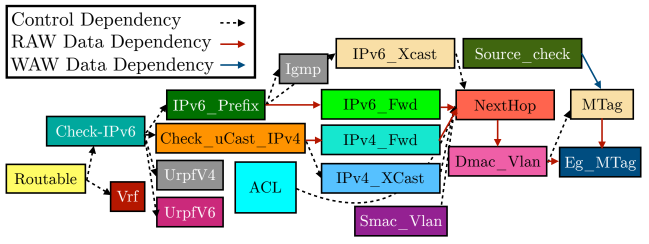

Figure 1: The TDG of the L2 L3 Simple MTag benchmark program implemented by Jose et al.

ASIC. Like traditional fixed-table ASICs, RMT processes packets with a linear pipeline of match action stages. Each stage contains a local SRAM and TCAM memory cluster to store custom match tables, and an action engine to carry out custom actions composed of a series of primitives.

3 PRECEDENCE

Precedence is a mechanism for resolving control and data dependencies between pipeline tables, allowing the tables to be placed on the same stage. It assigns every match-action rule a weighting, its precedence, which can either be a constant or the value of a metadata field. If multiple tables in a stage execute actions which store a value in the same metadata field, the conflict is resolved by storing the value of the action with the highest precedence. A rule's precedence field is treated similarly to any other table metadata field, and can be managed by runtime software.

Example: Consider the tables `Source_Check` (dark-green) and `MTag` (tan) shown in Figure 1. `Source_Check` always overwrites `MTag` if and only if it has an action to execute. In theory, the two tables could be placed in one stage by merging them into a single table, but in practice this is undesirable because the combined table requires a rule for every pair of rules in `Source_Check` and `MTag`; a combinatorial explosion of rules which consumes memory needlessly. Precedence allows both tables to share a stage by assigning `Source_Check`'s writes precedence 1 and `MTag`'s writes precedence 2. When both tables write, `MTag`'s write will take priority.

3.1 Architectural Model

Precedence is implemented as an extension of the RMT architecture by placing a new hardware component, the action output selector, between the action units' output and the outgoing metadata bus (Figure 2). The action output selector reads each action unit's output write and that output's precedence, which can either be sourced from a constant stored in the Very Long Instruction Word (VLIW) Memory, **or** a metadata field forwarded by the action input selector, **or** the output of another action unit. If multiple action units

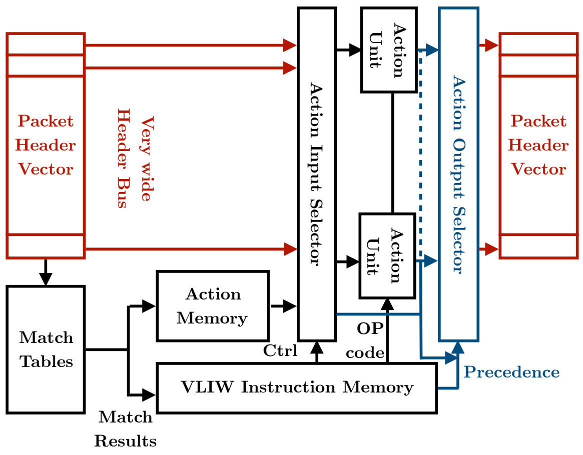

Figure 2: The RMT architectural model extended by precedence. New architecture added by the extension is colored in dark blue.

store their output in the same field the action output selector forwards the output with the highest precedence.

An action output selector for an n action unit RMT stage is constructed with a crossbar of n MUXes, each accepting each action unit's output as an input and sending their output to the outgoing metadata bus. Each MUX's SELECT is wired to a n-way comparator which can read an action unit's precedence from the output of another action unit, the action input selector, or the VLIW Instruction Memory.

3.2 Dependency Resolution

We now show how precedence can resolve the control and dataflow dependencies described in Section 2.

WAW data dependency resolution: WAW-dependencies, such as the `IPv4_UCast_LPM` example given above, can be resolved by giving each rule in the overwritten table a precedence of 1 and each rule in the overwriting table a precedence of 2. When the two tables are placed on the same stage, the overwriting table will overwrite the overwritten table.

Space Complexity: Any dependency between two tables can be naively resolved by merging the tables. In the worst case, merging two tables generates a rule for every pair of rules in the tables. If the two tables are denoted T_1 and T_2, and their rule number $|T_1|$ and $|T_2|$, then table merge generates $|T_1| \cdot |T_2|$ rules. By comparison, precedence resolves WAW-dependencies with no additional cost beyond the action output select, and thus only needs $|T_1| + |T_2|$ rules.

WAR data dependency resolution: Two tables linked by a WAR data dependency can be executed in a single stage.

RAW data dependency resolution: One might naively think that RAW data dependencies are intractable without merging. If table T_2 reads table T_1's output, one might reason, then T_1's output must be known before executing T_2. Consider, however, the case that T_1 performs a computation with a small number of outputs, like modulus or a conditional. Copies of T_2 for each possible outcome of t_1 could be executed speculatively in parallel, and precedence subsequently used to read T_1's result and select the execution of T_2 to store.

Example: Consider a data center routing protocol on an end-of-row switch indirectly connected to 3 core switches. To avoid directing all outgoing flows to a single core switch, the router computes mod 3 of each inbound packet's source IP and uses the result to choose between these three paths. This protocol could be implemented by two tables below:

```
t1: rnd_val <- src_IP % 3;
t2: next_hop <- route_tbl[dst_IP, rnd_val];
```

These tables can be placed in one stage using precedence by: **(a)** modifying T_1 to set three one bit flags `rnd_val_is_0`, `rnd_val_is_1`, and `rnd_val_is_2` according to `rnd_val`'s value, and **(b)** dividing T_2 into three tables which compute `route_tbl[dst_IP, 0]`, `route_tbl[dst_IP, 1]` and `route_tbl[dst_IP, 2]`, where `route_tbl[dst_IP, i]`'s output's precedence is `rnd_ele_is_i`. At runtime, T_2 is speculatively executed for all three possible values of `rnd_val` in parallel, and subsequently the action output selector looks at which `rnd_ele_is_i` flag is set to pick the correct version of T_2's output to store.

Space Complexity: To resolve a RAW-data dependency between two tables T_1 and T_2, precedence requires worst case $dom(T_1 \textit{ outputs read by } T_2) \cdot |T_2|$ rules, since one copy of T_2 needs to be made for every possible operand vector it could read from T_1. When $dom(T_1 \textit{ outputs read by } T_2)$ comparable to $|T_2|$, precedence's rule number is comparable to table merge (and requires many more action units). When $dom(T_1 \textit{ outputs read by } T_2)$ is a small constant, however, precedence only requires $O(|T_1| + |T_2|)$ rules.

In the previous example, merging the two tables in the code snippet above produces a single table with |`src_IP`| · |`dst_IP`| rules, which can be very large if |`src_IP`| and |`dst_IP`| are substantial. Resolving this dependency by precedence, however, only requires |`src_IP`| + 3|`dst_IP`| rules.

Control dependency resolution: To understand how precedence can resolve a control dependency, consider the pipeline shown in Fig 3. This pipeline has no dataflow dependencies, and each arrow indicates a control flow dependency. For example, tables T_1, T_2 and T_3 form a fully generic control flow dependency: after executing T_1, program control is either passed to T_2 or T_3. Despite needing to know T_1's output to determine whether to execute T_2 or T_3, all three tables can be placed in the same stage by speculatively executing T_2 and T_3 and using precedence to select which output to store.

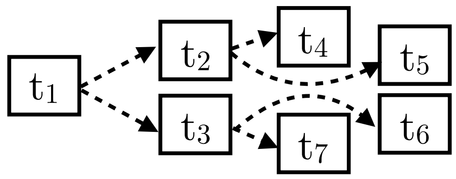

Figure 3: A pipeline with only control dependencies.

First, the control flow dependency can be converted into a data flow dependency by predication, replacing each of T_1's `jump(T2)` and `jump(T3)` actions with writes to the boolean predicate `is_jump_T2` and `is_jump_T3`. Next, T_2 and T_3's rules' precedence are set to their respective predicates. Finally, T_2 is instructed to write the special value `not_written` to any variable neither it nor T_1 originally wrote but T_3 did, and vice versa. At runtime, the action output selector can choose between T_2's and T_3's outputs because only one of `is_jump_T2` and `is_jump_T3` is set.

Space complexity: Precedence requires no additional rules to resolve control dependencies.

3.3 Combined Dependency Resolution

Not only can precedence resolve WAW, control flow, and certain RAW dependencies individually, it can also resolve any combination of these dependencies:

- To resolve a combined WAW and RAW dependency between two tables, T_1 and T_2, the precedence of T_1's rules is set to 0 so that its output will be overwritten by the selected speculatively executed copy of T_2.
- To resolve a combined WAW and control flow dependency between the parent table T_1 and the child tables T_2 and T_3, the parent table's rules are assigned precedence $\frac{1}{2}$, so that its output will overwrite its unchosen speculatively executed child and be overwritten by its chosen speculatively executed child.
- To resolve a combined RAW and control flow dependency between the parent table T_1 and the child tables T_2 and T_3, a copy of T_2 is made for each output of T_1 it could read, and similarly for T_3. Each predicate specifies both a table and an output of T_1.
- To resolve all three dependencies, proceed as the RAW & control flow case, but set T_1's precedence to $\frac{1}{2}$.

No change is required to resolve a WAR dependency in combination with any other set of dependencies. (To simply implementation, precedence values are scaled to be integers.)

3.4 Dependency Chain Resolution

Now that we have analyzed single dependencies, we turn to dependency chains. Consider a chain of WAW data dependencies $T_1 \longrightarrow T_2 \longrightarrow \ldots T_n$. Every table in such a chain can be executed in the same stage by assigning the i-th table's rules precedence i, so that T_i overwrites all tables before it in the chain and is overwritten by all tables after it.

Unfortunately, however, precedence cannot resolve chains of RAW data dependencies or control dependencies. Consider, for example, the pipeline TDG of control-dependencies shown in Figure 3. One might think that this pipeline could be executed in a single stage as we showed T_1, T_2 and T_3 could be previously by replacing T_2's jumps with the predicates `is_jump_T4` and `is_jump_T5`, and so forth. However, the value of `is_jump_T4` is not determined until the action output selector has processed T_2 and T_3's outputs; until then it could either be `true`, `false` or `not_written`. This means that T_4's precedence will not be known until the action output selector has chosen between the speculative executions of T_2 and T_3, so T_4 must be left to the next stage. In general, precedence cannot place tables in a 2 or more link RAW data/control dependency chain in the same stage.

3.5 Out-of-Order Execution

Thus far, precedence has only been used to place tables linked by a dependency in the same stage. Precedence can also be extended to enable out-of-order execution, where a table can be executed at any stage prior to a table it is dependent on.

If a speculatively executed table is placed in a stage prior to the table it depends on, its output is written to a set of temporary variables on the metadata bus. Then, in the stage containing the table it depends on, a small one rule table is added which reads each temporary variable and writes it, with its corresponding precedence, to the appropriate field. While this out-of-order execution strategy does take up an extra action unit for each out-of-order speculatively executed table, its memory overhead is minimal.

4 EVALUATIONS

We now evaluate precedence against a set of benchmark programs. We give the experimental setup (Section 4.1), and then measure the percentage of dependencies precedence resolves in the benchmark programs (Section 4.2). Next, we analyze its impact on the compiled depth of these programs (Section 4.3) and finally its hardware cost (Section 4.4).

4.1 Experimental Setup

Benchmark Programs. We benchmark on: **(1)** two real world, open source P4 programs, DC.p4 and switch.p4 and **(2)** four P4 programs used as benchmarks by Jose et al., L2 L3 Simple, L2 L3 Complex, L2 L3 Simple MTag and L3 DC. Table 1 gives key program attributes.

Target Switch Architecture. Programs are compiled to a precedence enabled RMT pipeline. The numeric values for this pipeline's attributes are chosen following Bosshart et al.'s recommendations in [4]. In particular, each match stage contains 106 SRAM blocks of 1K entries × 112b, 16 TCAM blocks of 2K entries × 40b, and a separate 640b crossbar for SRAM and TCAM, capable of querying 8 tables at once.

Program	Table Number	Maximum Table Size	Longest Dep. Chain
switch.p4	163	«1 stage	12
DC.p4	43	«1 stage	10
L2L3Simple	16	10 stages	6
L2L3Complex	25	4 stages	11
L2L3SimpleMTag	19	10 stages	8
L3DC	11	<1 stage	8

Table 1: Benchmark Program Table Number, Maximum Table Size, and Longest Dependency Chain.

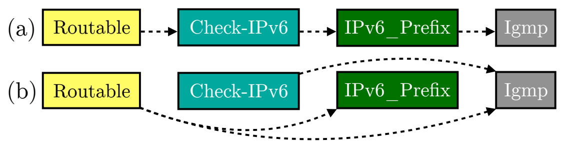

Figure 4: A control dependency chain in L2 L3 Simple MTag (a) without and (b) with precedence.

Compiler. The benchmark programs are compiled to the precedence enabled RMT pipeline using Jose et al.'s [13] compiler, after modifying it for precedence in two ways. **First**, all WAW-data dependency constrains are removed from each program's TDG. **Second**, for each chain of control dependencies in a program, each table's dependency on the next table in the chain is removed, and new dependencies between it and every other subsequent table in the chain added. For example, Figure 4 shows a chain of control dependencies from L2L3SimpleMTag without (Figure 4 (a)) and (Figure 4 (b)) with precedence. In (b), the dependency between Routable and Check_IPv6 (the next table in the chain) is removed and dependencies between IPv6_Prefix and Igmp (subsequent tables in the chain) added. All RAW data dependencies are assumed intractable unless otherwise stated.

4.2 Resolvable Dependency Frequency

First, we examine the percentage of dependencies in real-world routing programs that precedence can resolve. The number of RAW, WAW and control dependencies in each program are shown in Table 2. The programs are dominated by control and RAW dependencies. Precedence can reliably resolve every dependency except for RAW data dependencies. Figure 5 lists the number of resolvable dependencies in each program. Precedence can resolve a substantial minority of dependencies in all programs, and a majority in half. The more complex a program's control structure, the better precedence performs.

4.3 Pipeline Stage Reduction

Each benchmark program's pipeline depth with and without precedence is given in Figure 6. Precedence reduces the depth of DC.p4, switch.p4 and L3.DC by 40%, 48%, and 42%, while achieving more modest depth reductions on L2 L3 Simple, L2 L3 Simple MTg, and L2 L3 Complex (5%, 5% and 10%). Note

Program	RAW Deps.	WAW Deps.	Control Deps.
switch.p4	83	27	134
DC.p4	33	1	12
L2L3Simple	4	0	22
L2L3Complex	25	1	16
L2L3SimpleMTag	6	1	16
L3DC	7	3	1

Table 2: Benchmark Program Dependencies.

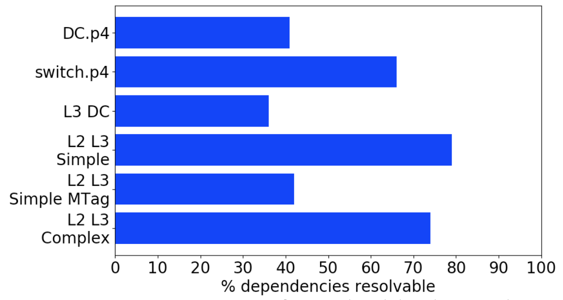

Figure 5: Percentages of resolvable dependencies.

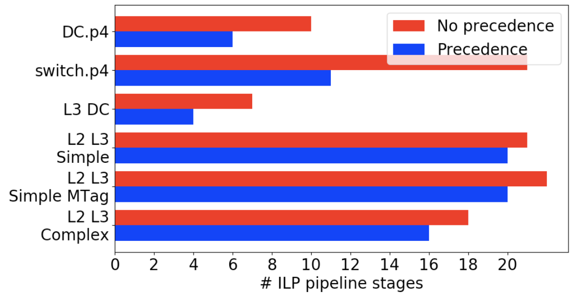

Figure 6: Layout depth decrease with precedence.

that switch.p4 requires double the match keys per stage for full depth reduction.

Precedence's performance difference between these two sets of programs is caused by their different structure. Figure 7 shows the layouts of DC.p4 and L2 L3 Simple without (left) and with (right) precedence. Each vertical column in Figure 7 represents a stage's memory, and each colored place in a column represent a table whose height represents the number of entries it contains. Two blocks with the same color on different stages indicate a table spanning multiple stages. Uncolored space indicates unused memory.

DC.p4 consists of 43 small tables joined by a complex web of dependencies. These dependencies limit table placement, resulting in significant unused space (Figure 7, DC.p4 (left)). By removing them, precedence recovers the space, achieving significant depth reduction (Figure 7, DC.p4 (right)). L2 L3 Simple's layout without precedence (Figure 7, L2 L3 Simple (left)), however, contains relatively very little unused space. Its depth is dominated by the size of its tables - in particular, two 160,000 entry tables (dark blue and grey) dominate 16 out of its 21 initial stages! Precedence's effects are more modest here because dependencies are not the main driver

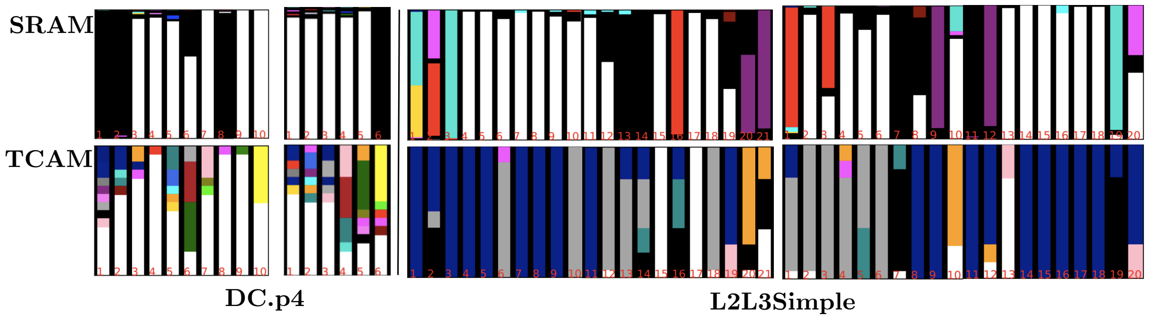

Figure 7: DC.p4 and L2L3Simple's layouts when compiled to a pipeline without (left) and with (right) precedence.

of pipeline depth. Notably, when the size of L2 L3 Simple's tables are halved, depth reduction increases from 5% to 20%. We argue, however, that future routing programs will look more like DC.p4 than L2 L3 Simple, using large numbers of (necessarily small) tables rather than a few monolithic tables to compute. Notably, precedence performs best on DC.p4 and switch.p4, the two real world programs.

DC.p4 also demonstrates the power of RAW dependencies resolution. DC.p4 contains two critical RAW dependencies: one on a table with four rules, and one on a table with nine. Even if no other control or WAW dependencies are removed, removing these RAW dependencies decreases DC.p4's layout to 6 stages. When tables are small, replication provides a practical way to remove RAW bottlenecks.

4.4 Hardware Cost

Finally, we evaluate the hardware cost of the output selector to show these benefits can be obtained feasibly. The output selector was modelled as a $224 \times 224 \times 19$b crossbar, connecting each of RMT's 224 ALU's outputs to the 224 fields on its 4Kb bus. The crossbar's wiring was synthesized with the ORION 3.0 [14] power and area simulator; its comparator logic is small and is neglected. ORION 3.0 synthesis gives a crossbar area of 0.95mm^2 using 54 nm process technology.

RMT stage logic (excluding memory) is estimated in [6] as 1.243mm^2 using 16 nm process technology. Scaling the synthesized area by 16/45 to match, the output selector adds 0.34mm^2 to each stage's logic, a 21% increase. Stage logic, however, is only a small fraction of RMT's chip area: the action engine is 7.4% of its area and its match crossbar is less. The output selector thus only increases RMT's area modestly.

5 RELATED WORK

Physical Layout Design. Many techniques have been proposed to optimize the datapath layout of a routing program. Jose et al. [13] and Dai et al. [7] give compilers to efficiently map a logical programming application into pipelined stages. Alternatively, Hardware Design Languages such as [18] and [24] allow programs to directly optimize datapath layout. Unlike these approaches, precedence's optimizations are primary hardware based and focus on dataplane architecture.

Switching ASIC Architecture Design. Several variants on RMT's switching ASIC architecture have been proposed. dRMT [6] disaggregates RMT's memory, placing it in a memory pool accessible over a centralized crossbar. dRMT removes most dependency constraints on table placement, but not on access, and requires a monolithic crossbar between its processors and memory. Banzai [22] equips each stage with more complex logic, potentially allowing more compact layout, but does not directly address dependency constrains. Many commercial switches, such as Cavium XPliant [5], Intel FlexPipe [12] and Broadcom Trident 3 [1], also use an augmented match-action pipeline architecture. While their precise architectures are not available, their capabilities are. Broadcom silicon, for example, can address WAW, WAR and control dependencies by combining the notion of logical tables with a concept similar to precedence.

Speculative Execution. Outside the scope of programmable switches, speculative execution [9, 16, 19, 21] is a well-studied method for achieving different granularities of parallelism. Further, the rise of general-purpose graph processing units (GPGPU) has led to the study of speculative parallelism in GPU-based systems. Diamos et al. [8] adopts thread-level speculation to automatically convert a sequential program into parallel execution blocks. Liu et al. [15] explores the benefits of using GPU to achieve software value prediction.

ACKNOWLEDGMENTS

The authors wish to acknowledge the contributions of Broadcom, Inc. to this paper in providing a conceptual and practical basis for precedence-based data-path and control-path execution in a (non-public) programmable switch ASIC pipeline, and further demonstrating how dependency resolution can be defined in a high-level network programming language. While this paper demonstrates how precedence can substantially augment the processing efficiency of a public switching architecture such as RMT, the authors acknowledge that commercial, non-public programmable switch ASIC architectures currently available or in development may leverage similar techniques to provide such advantages over conventional match-action pipelines.

REFERENCES

[1] Broadcom Trident 3. [n. d.]. XPliant Ethernet Switch Product Family. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56870-series/. Accessed: 2018-11-15.

[2] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Comput. Commun. Rev.* 44, 3 (July 2014), 87–95. https://doi.org/10.1145/2656877.2656890

[3] 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. In *ACM SIGCOMM Computer Communication Review*, Vol. 43. ACM, 99–110.

[4] 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. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM (SIGCOMM '13)*. ACM, New York, NY, USA, 99–110. https://doi.org/10.1145/2486001.2486011

[5] Cavium. [n. d.]. XPliant Ethernet Switch Product Family. https://www.cavium.com/xpliant-ethernet-switch-product-family.html. Accessed: 2018-11-15.

[6] 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*. ACM, 1–14.

[7] Jinquan Dai, Bo Huang, Long Li, and Luddy Harrison. 2005. Automatically Partitioning Packet Processing Applications for Pipelined Architectures. *SIGPLAN Not.* 40, 6 (June 2005), 237–248. https://doi.org/10.1145/1064978.1065039

[8] G. Diamos and S. Yalamanchili. 2010. Speculative execution on multi-GPU systems. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*. 1–12. https://doi.org/10.1109/IPDPS.2010.5470427

[9] Lance Hammond, Mark Willey, and Kunle Olukotun. 1998. Data speculation support for a chip multiprocessor. *ACM SIGOPS Operating Systems Review* 32, 5 (1998), 58–69.

[10] David Hancock and Jacobus van der Merwe. 2016. HyPer4: Using P4 to Virtualize the Programmable Data Plane. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies (CoNEXT '16)*. ACM, New York, NY, USA, 35–49. https://doi.org/10.1145/2999572.2999607

[11] Barefoot Inc. 2019. switch.p4. https://github.com/p4lang/switch/blob/master/p4src/switch.p4

[12] Intel. [n. d.]. Intel Ethernet Switch Silicon. https://www.intel.com/content/www/us/en/products/network-io/ethernet/switches.html. Accessed: 2018-11-15.

[13] Lavanya Jose, Lisa Yan, George Varghese, and Nick McKeown. 2015. Compiling Packet Programs to Reconfigurable Switches. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation (NSDI'15)*. USENIX Association, Berkeley, CA, USA, 103–115. http://dl.acm.org/citation.cfm?id=2789770.2789778

[14] Andrew B Kahng, Bill Lin, and Siddhartha Nath. 2012. Explicit modeling of control and data for improved NoC router estimation. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*. IEEE, 392–397.

[15] Shaoshan Liu, Christine Eisenbeis, and Jean-Luc Gaudiot. 2011. Value Prediction and Speculative Execution on GPU. *International Journal of Parallel Programming* 39, 5 (Oct. 2011), 533–552. https://doi.org/10.1007/s10766-010-0155-0

[16] Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, and Roger A. Bringmann. 1992. Effective Compiler Support for Predicated Execution Using the Hyperblock. In *Proceedings of the 25th Annual International Symposium on Microarchitecture (MICRO 25)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 45–54. http://dl.acm.org/citation.cfm?id=144953.144998

[17] J. Menon, M. de Kruijf, and K. Sankaralingam. 2012. iGPU: Exception support and speculative execution on GPUs. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*. 72–83. https://doi.org/10.1109/ISCA.2012.6237007

[18] Rishiyur Nikhil. 2004. Bluespec System Verilog: efficient, correct RTL from high level specifications. In *Formal Methods and Models for Co-Design, 2004. MEMOCODE'04. Proceedings. Second ACM and IEEE International Conference on*. IEEE, 69–70.

[19] Jeffrey T Oplinger, David L Heine, and Monica S Lam. 1999. In search of speculative thread-level parallelism. In *Parallel Architectures and Compilation Techniques, 1999. Proceedings. 1999 International Conference on*. IEEE, 303–313.

[20] David A. Patterson and John L. Hennessy. 1990. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[21] B Ramakrishna Rau and Joseph A Fisher. 1993. Instruction-level parallel processing: history, overview, and perspective. In *Instruction-Level Parallelism*. Springer, 9–50.

[22] 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*. ACM, 15–28.

[23] Anirudh Sivaraman, Changhoon Kim, Ramkumar Krishnamoorthy, Advait Dixit, and Mihai Budiu. 2015. Dc. p4: Programming the forwarding plane of a data-center switch. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*. ACM, 2.

[24] Donald Thomas and Philip Moorby. 2008. *The Verilog® Hardware Description Language*. Springer Science & Business Media.