Precedence: Enabling Compact Pipeline Layouts By Table Dependency Resolution

ABSTRACT

The emergence of programmable switching ASICs has substantially realized the goal of flexible, high speed, programmer friendly packet processing. A critical concern, however, when compiling a program into such an ASIC is the depth of the program's pipeline layout. The depth of a pipeline is correlated with its latency, which in turn is correlated with pipeline power consumption and FLOP count. Further, because many commercial pipelines such as RMT [3] and Flexpipe **??** cannot allocate memory freely to stages, non-compact pipelines can result in significant memory under-utilization in underloaded stages.

Inter-table control and data dependencies critically limit the ability of compilers to layout tables compactly, but how to design pipelines which can resolve these dependencies has not been studied before. In this paper, we introduce precedence, an extension of the RMT pipeline architecture which enables tables linked by dependencies to be executed in parallel or even out-of-order. We analyze precedence, and show that it resolves up to 90% of dependencies in our benchmark program suite, achieves pipeline depth reduction of up to 40%, and can be implemented with lightweight and inexpensive hardware.

1 INTRODUCTION

To achieve high-speed flexible packet processing, programmable switches have drawn a lot of attention from both academia and industry. Two architectures, Reconfigurable Match-Action Table (RMT) [4] and disaggregated Reconfigurable Match-Action Table (dRMT) [7], have been proposed to efficiently realize high-level programming languages such as P4 [2].

When a compiler realizes a high-level program into a programmable switch, however, it is critical to minimize the depth of the pipeline layout it generates. The length of a pipeline is strongly correlated with its latency, which in turn is strongly correlated with its power consumption and FLOP count. Moreover, most pipeline targets, such as RMT, either disallow or highly constrain stages from sharing memory. An unnecessarily long pipeline layout can lead to underutilization of each stage's resources, increasing its demand for costly SRAM and TCAM memory.

A critical limitation on the ability of compilers to generate compact pipelines are inter-table data and control hazards, or dependencies. Under current pipeline architectures, if, for example, a table, T_1, determines whether a second table, T_2, is executed, T_2 must be placed after T_1 in the pipeline's layout. As routing programs increasingly rely on complex systems of small, interconnected tables rather than a few, monolithic tables to perform computation, their data and control dependency constraints will become ever more cumbersome.

We therefore postulate that to allow compact, memory efficient pipeline layout, it is necessary for pipeline hardware to allow parallel or even-out-of-order execution of tables with data and control dependencies. In such an architecture, the only limitation on pipeline layout would be hardware constraints, allowing compilers to maximize each stage's resource utilization and minimize overall pipeline depth.

However, despite its great benefits, resolving inter-table control and data dependencies is not trivial because of the following challenges:

1. **Achieving correct parallel table execution:** Similar to what happens in multi-core or multi-thread systems, parallel execution of tables linked by control or data dependencies inevitably introduces data race conditions [19]. The architecture must handle these race conditions properly to guarantee correction execution of high-level programs.
2. **Achieving efficient parallel table execution in switching ASICs**. Programmable switches have a very high standard in terms of throughput, latency, and power consumption. Any ASIC that permits parallel execution of dependencies must do so while meeting the same performance criteria.

To the best of our knowledge, no programmable switching ASIC architecture that is made public has provided mechanisms to handle dependencies in their stage design. Meanwhile, while previous studies on speculative execution in CPU/GPU-based systems [9, 13, 15] have gained tremendous insights on handling race conditions in parallelism, their solutions cannot be directly applied in programmable switching ASIC architectures.

In this paper, we introduce *precedence*, an extension to the RMT hardware architecture which allows tables linked by data and control dependencies to be executed in parallel or even out-of-order via speculative execution. While the underlying idea of *precedence* is not new [18], we are the first to introduce it to the design of programmable switching ASICs. We analyze precedence and show that it resolves up to 90% of dependencies in our benchmark program suite,

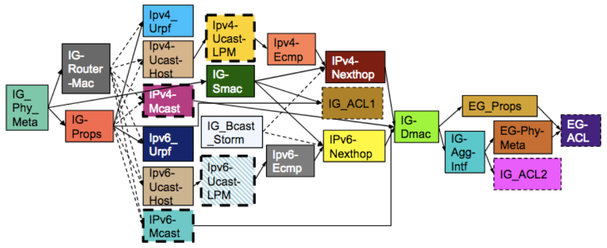

Figure 1: The TDG of the L2 L3 Complex benchmark program implemented by Jose et al.

achieves pipeline depth reduction of up to 40%, and can be implemented with lightweight and inexpensive hardware.

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 [12] 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 identified by CPU hardware designers. [12, 18] 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 Complex benchmark program implemented by Jose et al.

Since there may be multiple types of dependencies between two tables, a TDG may be a multigraph. We will show that if each dependency between two tables can be resolved individually, all dependencies between these tables can be resolved simultaneously.

2.2 Switch Architecture

We present precedence as an extension of the RMT switch architecture - a real, high-performance, programmable switch ASIC, as shown in Figure 2. RMT's architecture is archetypal of the growing class of programmable switch ASICs, supporting a DAG of match-action tables and equipped with an easily accessible metadata bus, easily accessible VLIW memory, and an array of per stage action computation units. Also notable are strong hardware constraints on the distribution of overall memory to stages, contributing to a need for flexible pipeline layout.

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 weight, or 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 `IPv4_UCast_LPM` and `IPv4_UCast_Host` shown in Figure 1. `IPv4_UCast_LPM` always overwrites `IPv4_UCast_Host` if and only if it has an action to execute. In theory, the two tables could be merged into one stage, but in practice this is undesirable because `IPv4_UCast_Host` can use cheap, less power intensive SRAM memory while `IPv4_UCast_LPM` must use expensive TCAM memory. Precedence allows both tables to share a stage by assigning `IPv4_UCast_Host`'s writes precedence 1 and `IPv4_UCast_LPM`'s writes precedence 2. When both tables write, `IPv4_UCast_LPM`'s higher priority write will take precedence.

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 VLIW Instruction Memory, OR a metadata field forwarded by the action input selector, OR the output of another action unit. If multiple action units 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 an array of n MUXes, each accepting each action unit's output as an input and sending their output to

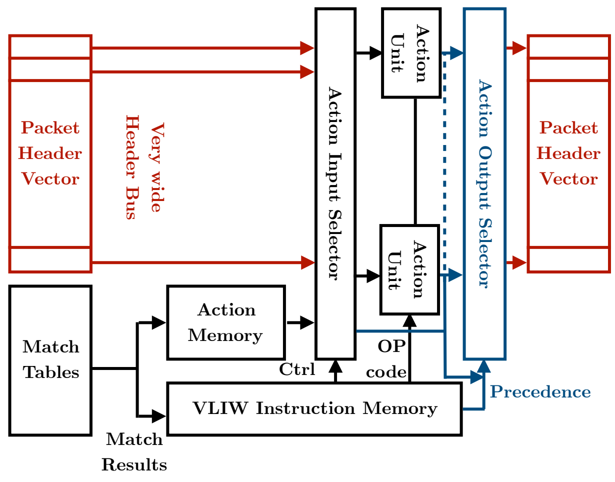

Figure 2: The RMT architectural model extended by precedence. New architecture added by the extension is colored in dark blue.

the outgoing metadata bus. Each MUX's SELECT is wired to a n-way comparator which can read an action unit's strength from the output of another action unit, the action input selector, or the VLIW Instruction Memory.

The block cost of the action output selector is linear in the number of action units and thus cheap. A 5V MUX has a typical propagation delay of 10ns and a 5.5V comparator has a propagation delay of 50ns, making the action output selector non-time intensive. The action output selector does, however, require a cross bar between its MUX and the action units, which has quadratic wiring complexity in the number of action units. To reduce this complexity when the number of action units is large, each MUX can only link to a constant subset of action units.

3.2 Dependency Resolution

Now that we have given a description of precedence and described its high-level architectural model, we proceed to show how precedence can resolve the control and dataflow dependencies described in Section 2.1.

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 overwritten table will only write to fields not written to by the overwriting 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, asymptotically better than table merge.

WAR data dependency resolution: Two tables linked by a WAR data dependency can be executed in a single stage without modification.

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 precendence subsequently used to read T_1's result and select the correct speculative execution of T_2 to store.

Example: As an example, consider a data center routing protocol on an end-of-row switch 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 srcIP and uses the result to choose between its three available paths. This protocol could be implemented by two tables executing the following instructions:

```
rnd_val = src_IP % 3;
next_hop = route_tbl[dst_IP, rnd_val];
```

These two tables can be placed in one stage using precedence by: **1)** 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 **2)** 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 the strength of `route_tbl[dst_IP, i]`'s output 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,

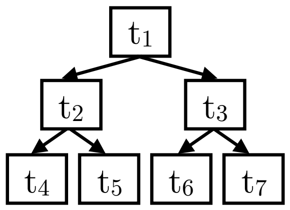

Figure 3: A pipeline with only control dependencies.

precedence only requires $O(|T_1| + |T_2|)$ rules and asymptotically beats table merge.

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 `t1`'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.

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, and thus performs asymptotically better than table merge.

3.3 Combined Dependency Resolution

Now that we have shown that precedence can resolve WAW, control flow, and some RAW dependencies individually, we proceed to show that precedence can resolve any combination of these dependencies linking two tables:

- To resolve a combined WAW and RAW dependency between two tables, T_1 and T_2, the strength 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 strength $\frac{1}{2}$, 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 strength to $\frac{1}{2}$.

No change is required to resolve a WAR dependency in combination with any other set of dependencies.

3.4 Dependency Chain Resolution

Now that we have analyzed dependencies between a parent table and a set of child tables, we finally turn to dependency chains. Consider a chain of WAW data dependencies $T_1 \rightarrow T_2 \rightarrow \dots T_n$. 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 examined as a mechanism to execute two tables linked by a dependency in the same stage. Importantly, however, precedence can 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, without additional hardware complexity.

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

Program	Description
L2L3 Simple	Simple L2 L3 program, large tables. [12]
L2L3 Complex	Enterprise DC aggregation router. [12]
L2L3 Simple MTag	Simple L2L3 with MTag [4] support. [12]
L3 DC	L3 routing, small enterprise switch. [12]
Local L2	L2 routing, small enterprise switch.
Interdomain	Inter-AS routing.

Table 1: Evaluated benchmark programs.

is added which reads each temporary variable and writes it, with its corresponding strength, 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

Now that we have presented precedence, we analyze its performance on a set of benchmark programs. First, in Section 4.1, we analyze precedence's applicability by measuring the percentage of dependencies in these programs it can resolve. Next, in Section 4.2, we analyze its impact on the compiled depth of these programs.

4.1 Resolvable Dependency Frequency

We begin our analysis of precedence by examining the percentage of dependencies in real-world routing programs that precedence can resolve. Our suite of benchmark programs, listed in Table 1, consists of **1)** the four P4 programs used as benchmarks by Jose et al. [12], L2 L3 Simple, L2 L3 Complex, L2 L3 Simple MTag and L3 DC, and **2)** two functional programs, Local L2 and Interdomain, which examine L2 enterprise and inter-autonomous-system routing use cases.

The number of dependencies in each dependency class were tallied for each program, and the results are shown in Figure 4. Note that the tally differentiates between tractable RAW data dependencies, where the domain of the shared variable is small enough for speculative execution, and intractable RAW dependencies, where this isn't true.

With the exception of Interdomain benchmark program's dependencies were dominated by control and RAW dependencies. This makes intuitive sense: any transmission of information between tables generates a control or RAW dependency, whilst WAW dependencies, which do not transmit information, were generally limited to pipeline optimization (e.g. splitting a table between SRAM and TCAM memory.) Because Jose et al. did not provide the contents of their P4 programs, we made the worse case assumption that all RAW dependencies were intractable (although some dependencies, like RAW to a packet's VLAN tag, conceivably might not have been.) Interdomain, however, did contain tractable

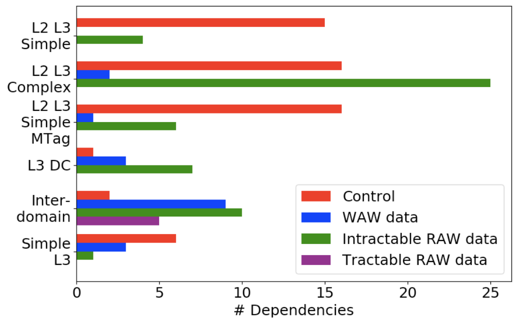

Figure 4: Number of each dependency class in typical programs.

L2 L3 Simple	L2 L3 Complex	L2 L3 Simple MTag
0.79	0.42	0.74
L3 DC	**L3 Local**	**Interdomain**
0.36	0.9	0.58

Table 2: Percentage of resolvable dependencies.

RAW dependencies, largely on the fields srciSP and dstISP (which referred to one of a small list of adjacent ISPs).

Precedence can resolve every dependency except for intractable RW dependencies. Table 2 lists the percentage of resolvable dependencies in each program. In every program, precedence could resolve a substantial minority of dependencies and in 4 out of 6 programs precedence could resolve a majority of dependencies. The applicability of precedence to a program depends on the complexity of the information flowing between tables. In programs like L2 L3 Simple, where inter-table communication is dominated by one table determining whether another executes precedence can address more than 3/4 of all dependencies, while in programs like L3 DC, where inter-table communication is dominated by the transmission of highly-complex variables, precedence can only address 1/3 of all dependencies.

4.2 Pipeline Stage Reduction

Next, we analyzed the impact of strength on pipeline depth by running Jose et al.'s ILP program compiler set to minimize pipeline depth on three of the benchmark P4 programs: L2 L3 Simple, L2 L3 Complex and L3 DC on an RMT pipeline with and without precedence, and recorded the reduction in pipeline depth.

The ILP pipeline compiler was modified to compile to a precedence enabled RMT pipeline by removing each program's single WAW data and control dependency constraints. As before, all RAW dependencies were assumed to be intractable and their dependencies were left untouched. Other than that, no chances were made to the benchmark programs.The benchmark program tables were large - L2 L3 Simple's tables had a median of 16,000 entries and ranged up to 160,000 entries - so up to 2 single entry tables required to execute tables with a control dependency out-of-order were neglected. The pipeline depths of the compiled benchmark programs with and without precedence are given in Figure 5.

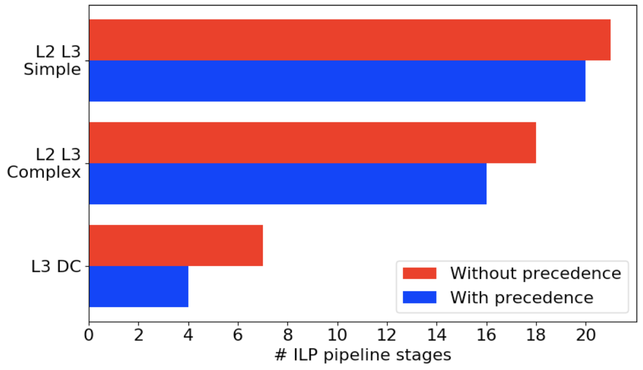

Figure 5: Pipeline depths of benchmark programs with and without precedence.

While precedence achieved a 40% reduction on L3 DC's pipeline depth, it reduced L2 L3 Simple and L2 L3 Complex's pipeline depths by a more modest 5% and 10%. This difference in performance can be understood by comparing L3 DC's compiled layout (Figure 6) with L2 L3 Simple's (Figure 7).

(*Note:* the layouts indicate each table contained in TCAM or SRAM memory by an uniquely colored block with height equal to the number of entries in that stage. Two blocks with the same color on different stages indicate a table spanning multiple stages. Uncolored space indicates unused memory.)

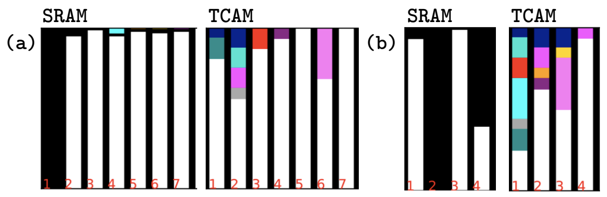

Figure 6: The L3 DC program's layout when compiled by ILP to (a) a standard RMT pipeline and (b) a precedence-enabled RMT pipeline.

Examining DC L3's layout on a RMT pipeline without precedence (Figure 6 (a)), it is clear that each stage contains significant unused memory. DC L3's depth is dominated by program dependencies, and by removing them precedence achieves substantial depth reduction by allowing this space to be recovered (Figure 6 (b)). L2 L3 Simple's layout without precedence (Figure 7 (a)), however, contains relatively very little unused space. Its depth is dominated by the size of its tables - in particular, its 160,000 entry tables (dark blue and grey) dominate 16 out of its 21 initial stages! When the size of L2 L3 Simple's tables are halved, precedence's depth reduction increases from 5% to 20%. Dependency resolution is simply ineffective when dependencies are not the main constraint on pipeline layout, explaining precedence's more modest effect.

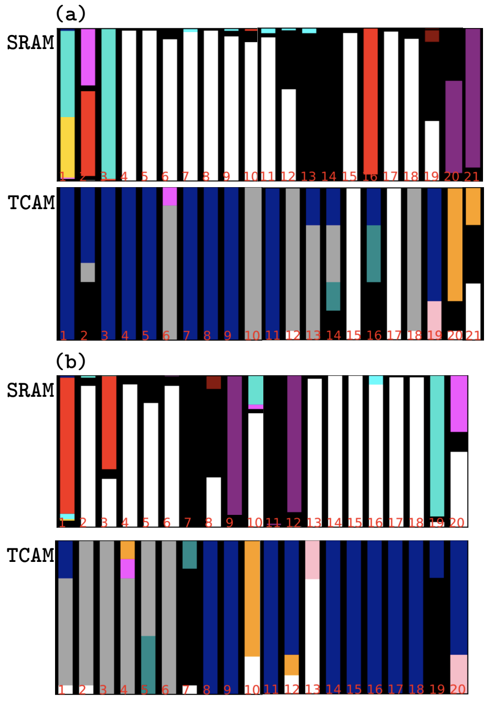

Figure 7: The L2 L3 Simple program's layout when compiled by ILP to (a) a standard RMT pipeline and (b) a precedence-enabled RMT pipeline.

We believe, however, that future of routing programs will be look more like L3 DC than L2 L3 Simple, relying more on complex, incremental computation than single monolithic tables. In this future, precedence's ability to resolve complex dependence graphs will greatly improve pipeline layout flexibility.

5 RELATED WORK

Physical Layout Designing A programmable switch may realize the same logical program with a range of physical layouts. The physical layout that the compiler designs has great impact on the switch's performance. As such, there are a great body of other work in addition to our paper which attempts to optimize the hardware realization of routing programs. Most signficiantly, Jose et al. [12], whose work was heavily used in this paper, designed an ILP solver to autonomously generate pipeline layout, while Dai et al. [8] map a sequential packet processing application into pipelined stages.

An alternative choice against the compilers is Hardware Design Languages (HDL) like [21], [16], which allows programmers to directly design physical layout and provides programmers great flexibility and optimization opportunities.

Programmable Switch Architecture Extensions There have been many proposed extensions to the RMT architecture. For example, dRMT [6] processes each packet with a dedicated core which can access all data originally separated among stages. It avoids the performance cliff when a switch does not have a sufficient number of stages, and its multicore designs can also benefit many existing works on GPU work. Another example is Banzai [20] switch, which extends the match action table to allow more complex instructions. It takes the expanse of using more space for logic units (e.g., ALUs) in a single stage, and thus can implement the same functionality with fewer stages.

There are also commercial switches like Cavium XPliant [5], Intel FlexPipe [11] and Broadcom Trident 3 [1] using different architectures. Although their detailed designs are not open, we can recognize their benefits through the features they claimed. For example, FlexPipe can resolve WAW dependency inside a single stage, which is quite similar to our precedence.

Speculative Execution in Hardware-Accelerated Systems. Outside the scope of programmable switches, speculative execution [10, 14, 17, 19] is a well-studied method to achieve different granularity of parallelism. With the rise of general-purpose graph processing units (GPGPU), speculative parallelism in GPU-based systems also draws a lot of attention. Diamos *et.al.* [9] adopts thread-level speculation to automatically convert a sequential program into parallel execution blocks, and applies optimizations to fully utilize CPU+GPU systems. Liu *et.al.* [13] explores the benefits of using GPU to achieve software value prediction. iGPU [15] breaks GPU code into idempotent regions and extends the GPU architecture to support exceptions, fast context switching and speculation using these regions as execution units.

REFERENCES

[1] Broadcom Trident 3. [n. d.]. XPliant Ethernet Switch Product Family. https://packetpushers.net/broadcom-trident3-programmable-varied-volume/. 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] Sharad Chole, Andy Fingerhut, Sha Ma, Anirudh Sivaraman, Shay Vargaftik, Alon Berger, Gal Mendelson, Mohammad Alizadeh, Shang-Tse Chuang, Isaac Keslassy, Ariel Orda, and Tom Edsall. 2017. dRMT: Disaggregated Programmable Switching. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17).* ACM, New York, NY, USA, 1–14. https://doi.org/10.1145/3098822.3098823

[8] 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

[9] 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

[10] 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.

[11] 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.

[12] 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

[13] 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

[14] 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

[15] 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

[16] 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.

[17] 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.

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

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

[20] 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.

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