Stitch - a DSL for active networking

Danny Yang (dzy4)

12/12/19

Source: https://github.com/yangdanny97/p4-stacklang

1 Introduction

Active networking is a networking paradigm that allows user-defined computations carried in packets to be executed on the network, allowing for dynamic modification of the network's behavior. Stitch is a domain specific bytecode language that implements a form of active networking. Packets containing Stitch programs can be executed by switches, allowing arbitrary computations and modifications to the network in real-time. This enables a wide variety of measurements, computations, and state changes to be performed without modifying/recompiling the switch. The name Stitch is a portmanteau of the words "stack" and "switch".

2 Design & Implementation

Stitch is implemented in p4 as a single file that contains the specification for a switch. There are two main additions: custom headers (that contain execution metadata, instructions, and a pre-allocated stack), and an interpreter for Stitch instructions which can execute a single instruction each time a packet passes through. The target-specific details of this implementation are based on the bmv2 simple_switch/v1model target.

2.1 Packet Format

To support executing Stitch programs, packets need to be augmented with a metadata header, a program data header, a pre-allocated stack that is used during execution, and a list of instructions. The main data type in Stitch is a 32 bit integer (interpreted as signed or unsigned depending on context). In the example implementation, the total size overhead of these augmentations is 326 bytes; an instruction is 5 bytes (1 byte for the opcode, 4 bytes for the value) and each stack position holds 1 value. However, the maximum number of instructions and pre-allocated stack slots can be configured to change this overhead. Smaller program and stack sizes can decrease the packet size overhead, but this potentially limits the complexity of programs. The number of instructions and the number of stack positions should not sum up to more than 250 to avoid splitting the packet.

Metadata Header
The fields of the metadata header will be target-specific - they hold metadata values which are updated after regular ingress parsing when the packet first arrives at a switch. When the `metadata` instruction is executed, the values are read from this header for reasons which I will

discuss in section 2.3. The metadata in the example implementation is designed for the v1model target, and has the following fields that can be accessed by Stitch programs:

Field Name	**P4 Type**	**Description**
ingress_port	bit<9>	the number of the ingress port on which the packet arrived to the switch
packet_length	bit<32>	the size of the packet in bytes
enq_qdepth	bit<19>	the depth of the queue when the packet was first enqueued
deq_qdepth	bit<19>	the depth of the queue when the packet was first dequeued
egress_spec	bit<9>	the egress port assigned before any Stitch instructions are executed
enq_timestamp	bit<32>	global timestamp in ms when the packet is first enqueued at this switch (recirculating to run additional instructions will not update this)
deq_timedelta	bit<32>	time in ms it took for the packet to pass through the switch, updated each time it recirculates
enq_qdepth	bit<32>	queue depth at the time the packet was enqueued for egress, updated each time it recirculates
deq_qdepth	bit<32>	queue depth at the time the packet was dequeued for egress, updated each time it recirculates
egress_timestamp	bit<32>	global timestamp in ms when the packet reached egress processing, updated each time it recirculates
switch_id	bit<32>	defaults to 0, unique ID for each switch assigned by the controller
rx_util	bit<32>	number of bytes received in the same ingress port as the program, since the last time it was read (reading this field also pushes the time in ms since the last time the rx_util from that port was read, and resets the bytes counter)
tx_util	bit<32>	number of bytes received in the same egress port as the program, since the last time it was read (reading this field also pushes the time in ms since the last time the tx_util from that port was read, and resets the bytes counter)

Program Data Header

The fields of the program data header are outlined below:

Field Name	**p4 Type**	**Description**
pc	bit<32>	the index of the current instruction, initialized to 0 and reset after each execution.
sp	bit<32>	the index of the **next** empty stack index, initialized to 0
steps	bit<32>	the number of instructions that have been executed in this run of the program, initialized to 0 and reset after each execution
done_flg	bit<1>	1 indicates the program is finished executing, initialized to 0 and reset after each execution
err_flg	bit<1>	1 indicates the program has encountered an error, initialized to 0
padding	bit<6>	

result	int<32>	stores the result of the program's execution (the value at the top of the stack when a `setresult` instruction is executed)
curr_instr_opcode	bit<8>	the opcode of the current instruction
curr_instr_arg	int<32>	this field **is not guaranteed** to store the operand of the current instruction

Stack and Instruction Headers
The list of instructions is implemented as a stack of 32 instruction headers. Instructions consist of an opcode (1 byte), as well as a 32-bit integer operand which is set to 0 for instructions with no operands. There will always be 32 instructions in each program; shorter programs are padded with `error` instructions which helps detect incorrect jumps. Then a `last` instruction (which behaves like `error`) is appended to delimit the instructions and the stack.

Field Name	**p4 Type**	**Description**
opcode	bit<8>	the opcode of the instruction
arg	int<32>	The operand of the instruction, set to 0 for instructions that do not use operands.

The stack is implemented as a stack of 32 stack headers, each of which holds a 32-bit integer.

Field Name	**p4 Type**	**Description**
value	int<32>	The value in this position of the stack, initialized to 0.

2.2 Instruction Set

The instruction set is inspired by bytecodes like JVM and Python, but also includes some stack operations available in languages like Forth. The size of the instruction set is kept relatively low for simplicity (right now there are around 3 dozen instructions). The opcodes are 8 bits, which allows for up to 256 instructions. Individual instructions are kept simple and have 0-1 operands, which makes parsing them very straightforward. Most instructions that implement redundant functionality, like JVM's `dup2` are omitted at the expense of having slightly larger program sizes.

Instruction	**Description**
`load [n]`	copy the value at offset [n] from the bottom of the stack, and push it to the top of the stack
`store [n]`	pop the value at the top of stack and store it at offset [n] from the bottom of the stack
`loadreg [n]`	copy the value in register [n] on the switch, and push it to the top of the stack
`storereg [n]`	pop the value at the top of stack and store it in register [n] on the switch
`push [n]`	push [n] on top of stack
`drop`	pops the value on top of the stack and throws it away

`add/mul/sub/sal/sar`	the left operand is the top of the stack, the right operand is the second from the top, pops both operands and pushes the result onto the stack.
`neg`	unary integer negation
`reset`	set SP to 0, essentially dropping the entire stack; used if the stack should not be preserved between hops
`and/or`	values > 0 are treated as truthy and values <= 0 are treated as falsy; pushes 1 if result is true, 0 if false
`not`	unary boolean negation (values > 0 get turned into 0, values <= 0 get turned into 1)
`gt/lt/gte/lte/eq/neq`	pushes 1 if result is true, 0 if false
`dup`	make a copy of the top value on the stack and push it onto the stack
`swap`	swaps the top 2 values in the stack
`over`	make a copy of the second value from the top the stack and push it onto the stack
`rot`	rotate the top 3 values on the stack, such that the 3rd from the top becomes the top, and the top 2 values move down
`jump [n]`	set PC to [n]
`cjump [n]`	pop the value on top of the stack, if it's truthy then jump to [n] otherwise fall through
`done`	set the [done] flag to 1
`error`	set the [error] flag to 1
`nop`	does nothing
`metadata [n]`	push the value of some switch/packet metadata field [n] to top of stack; values are extended/truncated to fit; available fields are depend on target architecture; >1 value may be pushed depending on the metadata field, and side effects are possible (for example, tx_util and rx_util also push the time since the last probe, in addition to resetting the counters for that value)
`setegress`	pop top of stack and set egress spec to the port corresponding to that value; this ends the current execution of the program, and will emit the packet out of the specified port and reset the PC/steps fields in the program data header
`setresult`	pop the top of stack and puts the value in the result field of the program data
`varload/varloadreg`	variant of load/loadreg which pops the top of the stack and uses that value as the offset/register number to read; this effectively replaces the top value on the stack, and the size of the stack should not change
`varstore/varstorereg`	variant of load/loadreg which pops the top 2 values of the stack; the top value is used as the offset/register number to write to, and the second value is the value that is stored

2.3 Program Execution

Stitch instructions are executed during ingress and egress processing. First, the program data header is checked to see if a stopping condition is met:

- If the done flag is nonzero, then the PC, steps, and done flag are reset and the packet is forwarded according to the forwarding table.
- If the error flag is nonzero, then the packet is forwarded without resetting the flag or modifying any fields, to preserve the error state for the end host to inspect, and to prevent the program from being executed on subsequent hops.
- If the step count exceeds the maximum, then the error flag is set and the packet is handled like the previous case. The limit on the step count exists to prevent infinite loops, but may sometimes prevent complex programs from fully executing.

If no stopping condition is met, then it means that execution can proceed, and the egress port is set so that the packet will be forwarded back to the current switch (potentially overridden if the `setegress` instruction is run). The stack and instructions are extracted from their header stacks into registers and the current instruction is read into the program data header and executed. Then, updated stack values are written back into the header stack. The reason for transferring all the stack values to a set of registers and back is for ease of implementation (on the bmv2 software switch, registers allow non-constant indexing but header stacks do not).

Only a single instruction can be executed at a time, so to execute the next instruction, the switch forwards the program back to itself. This is done using a user-defined port number in the topology, as opposed to a special recirculation port, because the v1model target does not support recirculation as defined in the P4-16 PSA. On each hop of the path a program-containing packet takes to get from the source to the destination, the entire program will execute from the beginning until a stopping condition is met, before being forwarded to the next hop.

Some metadata fields are only available during egress processing (in the example implementation, `enq_timestamp`, `deq_timedelta`, `enq_qdepth`, `deq_qdepth`, `egress_timestamp`, and `tx_util`). For those fields, during ingress processing the `metadata` instruction will push a blank value onto the stack, and the correct value will be pushed during egress processing. Since only a single instruction is run each time a packet traverses the switch, this does not cause any issues.

Forwarding Override
The only other way for a program execution to terminate is the `setegress` instruction. This instruction overrides the egress port metadata field and resets the PC, steps, and done fields in the program data header. In this case, the packet will not be forwarded back to the same switch and instead it is emitted from the specified port (or dropped, if the specified port corresponds to the special DROP_PORT).

Memory Abstractions
Stitch programs can access two types of memory: registers and the stack. Registers will maintain state between different programs/packets and are not shared between switches. The stack is preserved between hops. However, it is possible to write programs which discard the stack after each hop by having `reset` as the first instruction of the program.

Reading Metadata
The `metadata` instruction allows stitch programs to read switch/packet-related metadata (fields available will depend on the target). Due to the need for packets to be forwarded back to the same switch to execute multiple instructions, the metadata values when the `metadata` instruction is executed may have changed from when the program first arrived at the switch. For example, `ingress_port` will be 5 for each instruction besides the first. Thus, when a packet first arrives at a switch, these metadata values are copied into the metadata header to preserve them for use by later instructions.

3 Applications

Stitch is expressive enough for a wide variety of applications, although currently the complexity of programs is limited by the maximum stack, instruction, and step counts. It's possible to write programs that execute on each hop or programs that only execute on specific hops. Programs can also read switch metadata, override packet routing behavior, and can choose to either preserve or reset the stack between hops. As a proof of concept, I have developed several test programs (encoded using Scapy in Python) that demonstrate the capabilities of this language. They are available in the repository for this project.

Computation
I implemented several toy programs which calculate factorial and fibonacci. There are two different versions of the factorial program, one of which uses the stack and the other of which uses registers on the switch.

Source Routing
Source routing is trivial to implement in Stitch. The program is a single `set_egress` instruction, and the stack is initialized with the desired output ports in reverse order (with the first hop on top).

Packet Drop Detector
For this application, I wrote 3 programs which use a register on the switch to count the number of received packets. The first program causes the packet to be dropped by the first switch that receives it (thus only incrementing the counter on that switch). The second program represents a regular packet, and increments the counter of every switch on its path. The third program counts and returns the minimum counter value on its path, which represents the number of packets that successfully arrived at the destination.

I also wrote another program that calculates the difference between the minimum and maximum counter values on its path; although this is not a good metric for detecting dropped packets, it may have use in a different application.

Match-action Table

Stitch programs can be used to simulate a match-action table using registers, which proves that they can modify and define network behavior. This overcomes a limitation in p4, which does not allow modification of match-action table entries by the data plane.

First, a Stitch program populates registers on each switch with the forwarding table entries, such that a table entry that matches destination [i] to egress port [j] will correspond to register [i] containing the value [j].

Second, packets that want to use the table are sent with their desired destination initialized on the stack, and the Stitch program on the packet reads the corresponding register and sets the egress port accordingly.

Other Applications
With sufficient support from the controller, this system can support applications that make use of queueing and utilization metadata. This means that sophisticated forms of load balancing, congestion control, and telemetry are possible. Link utilization (as a moving weighted average) is not provided directly, but can be calculated from the provided metadata by regularly polling each switch. The repository for Stitch includes some example programs which read link utilization values, but there is no example of a controller that uses those values.

4 Related Work

This project is inspired by the Tiny Packet Programs paper, as well as two past projects from this course: TPP* and Tiny RISC-V.

Tiny Packet Programs
TPP is similar to Stitch in that instructions and a pre-allocated stack are stored with each packet. The practical difference is that TPP programs have less overhead and execute much faster, but Stitch supports longer programs and more complex computations thanks to its larger instruction set. TPP can have a variable number of instructions/stack size, while Stitch's stack size and instruction count are fixed. Compared to Stitch, TPP has more/easier access to metadata and information about switches and links. TPP also has an additional addressing mode based on hops, which is not supported in Stitch.

The runtime details are not exactly the same due to the different execution models (TPP executes all the instructions in one pass and cannot jump backwards, whereas Stitch only executes one instruction at a time), but both TPP and Stitch provide guarantees that instructions will be executed sequentially.

TPP*
TPP* is a partial p4 implementation of TPP by students in a past iteration of this course. It features an added instruction, `CMPEXEC`, which is a more flexible version of the `CEXEC` instruction. The

stack size and instruction count for TPP* are much lower than for Stitch, and it cannot access metadata fields at all. Another major difference (not by design) is that TPP* is implemented as a p4 module while the p4 implementation for Stitch is integrated with the switch.

TPP* programs can be translated to Stitch using the translation below, which I used for the packet drop detector example in the TPP* repository.

TPP	**Stitch**
`PUSH [from_reg]`	`loadreg [from_reg]`
`LOAD [from_reg] [to_offset]`	`loadreg [from_reg]; store [to_offset]`
`STORE [to_reg] [from_offset]`	`load [from_offset]; storereg [to_reg]`
`POP [to_reg]`	`storereg [to_reg]`

`CSTORE [to_reg] [old] [new]`	`load [old]`
	`loadreg [to_reg]`
	`neq`
	`cjump <idx of next_instruction>`
	`load [new]`
	`storereg [to_reg]`
	`<next_instruction>`

`CEXEC [reg] [offset]`
 	
Note: `CMPEXEC` translation is the same, but with a different comparison op
	`loadreg [reg]`
	`load [offset]`
	`eq`
	`cjump <idx of next_instruction>`
	`next_instruction`

Tiny RISC-V
Tiny RISC-V is another project by students in a past iteration of this course. It shares a similar execution model to Stitch (where a single instruction is executed at a time), but the overall goals of the projects differ. Both support a form of in-network computing, but Tiny RISC-V cannot read or use packet metadata and only works on a specific load-balancing topology, so it is not useful for network diagnostic tasks.

Tiny RISC-V's instruction set seeks to implement a subset of RISC-V, so it is quite different from Stitch. Tiny RISC-V has larger instructions with more operands, and has no stack. Instead, it uses on-packet registers and a heap that is shared between switches. In contrast, Stitch uses an on-packet stack, and registers are used to maintain state on each switch; there is no shared heap.

Internally, Stitch's implementation of the stack is based off of Tiny RISC-V's implementation of registers; both are represented as header stacks on the packet, but are parsed into registers when instructions are executed.

5 Conclusion

Overall, Stitch seems to be expressive enough to implement a wide variety of programs that can gather information and modify the network's behavior. This comes at the cost of execution speed and a pretty large overhead on packet sizes, despite having limits on program complexity due to the limits on instruction count/stack size/steps. Decreasing these limits will reduce the maximum possible execution time and the packet size overhead at the expense of further limiting the complexity of programs, but I think that it's good to at least have this choice.

Throughout the implementation process, I ran into several limitations and bugs with p4 and p4c, which I eventually overcame. The biggest challenge was reading from and writing to the stack. The bmv2 software switch did not allow non-constant indexing of header stacks, and switch statements were not allowed in the ingress stage; this led me to try to use if/else in place of a switch statement to determine which read statement to execute, but this caused massively increased compile times and led the compiler to crash during an optimization stage. I also attempted to use a table that matched on the read/write index, but p4c did not allow me to conditionally apply tables inside actions. The solution I ultimately decided on was based on a past project from this course, and involved reading the entire stack into registers, operating on it, then writing it back into the header stack.

Due to the execution model, Stitch programs that use registers on the switch are unreliable if multiple programs are running concurrently on the network. This is because programs are not executed all at once, and instructions from multiple programs that operate on the same set of registers may be interleaved. This represents a significant issue and it is worth investigating alternative execution models that can preserve the current flexibility while maintaining the illusion of atomic program execution.

References

Vimalkumar Jeyakumar, Mohammad Alizadeh, Yilong Geng, Changhoon Kim, David Mazières, "Millions of Little Minions: Using Packets for Low Latency Network Programming and Visibility (Extended Version)," arXiv:1405.7143 [cs], May 2014.

Danny Qiu and Chesley Tan, "Distributed Computation in p4," GitHub repository, https://github.com/dannyqiu/riscv-switch, Dec. 2018.

Peter Li and Tyler Ishikawa, "Tiny Packet Programs as a p4-backed DSL," Github repository, https://github.com/peteli3/p4-tiny-packet-programs, Dec. 2018.