CS 6114 Final Project Proposal

Chesley Tan (ct353), Danny Qiu (dq29)

November 2018

1 Introduction

We propose building a distributed computing system in P4 modeling the execution of a multi-processor computer on a network of switches running P4. This distributed computing system leverages an instruction-based execution model with an assembly-like language (referred to simply as "assembly" in the following system description) that supports operations for performing computations on operands in registers, loading/storing data into memory, and branching based on conditions. To provide the abstraction of "unlimited" memory on the programmable switches, we use an external datastore with two-way communication with the switches. This external datastore is implemented as a standalone Python program implementing a server that stores arbitrary information in memory and responds to packets requesting read/write operations to memory.

2 Architecture

To the use the distributed computing system, a client sends a packet with custom headers encoding an "assembly" program to a load-balancer (a P4 programmable switch, perhaps uniquely identified within the network using a reserved IPv4 address). The load-balancer acts as a gateway to the P4 switch network. The load-balancer then wraps the client's program packet in a header that contains metadata about the source of the program (the identity of the client making the request), the number of execution steps of the program (to prevent denial of service from long-running programs), and assigns a memory namespace ID to the program (uniquely determined by the client's identity). The load-balancer then forwards this packet to one of N execution units, each one a P4 programmable switch that implements logic to evaluate the "assembly" programs.

All "assembly" programs begin with a prologue that contains the register contents corresponding to the current execution state of the program. These registers contain data used as operands in "assembly" instructions as well as a special program counter register that keeps track of the next instruction to execute. The register values are encoded into the packet as separate headers,

one corresponding to each register value. The instructions in the program itself are encoded as separate headers, one for each instruction. This allows us to retrieve the next instruction to execute by indexing into a parsed array of program instruction headers using the value of the program counter from the program counter header. Whenever the program makes a load/store from memory, the switch forwards the program packet to the datastore with an additional header signifying whether a load or store operation is being performed and on what memory address. The datastore responds to packets requesting a load operation by forwarding the received program packet back to the execution unit which sent it, but with an additional header containing the loaded data value from memory within the client's previously assigned memory namespace. The datastore responds to packets requesting a store operation by performing the memory update and forwarding the received program packet pack to the originating execution unit.

The execution units are responsible for executing the "assembly" programs one instruction at a time. After completing an instruction, the execution unit increments the program counter register and forwards the program back to itself. This naturally simulates multiprogramming, as a single execution unit can simultaneously execute multiple programs, with the packet scheduler acting as a thread scheduler. When the program execution is complete (by either the program counter's reaching the end of the program, or by reaching a maximum number of execution steps), the results (the value of all the registers) are forwarded back to the load-balancer, which parses the packet header to determine the client that requested the program and forwards the results back to the appropriate client.

3 Considerations

One important limitation of this system is that the network is unreliable and packet buffers could fill up if there are many simultaneous programs being executed. Dropping of packets within the network would cause the program to fail to completely execute. An inefficient, but potential workaround is to require the client to timeout when a response is not returned within a given timeframe and re-transmit the program with timeouts until a response is finally returned. However, the partial execution of programs may leave memory in an unpredictable state, so clients would have to write "assembly" programs that can tolerate this form of failure.

Another important limitation is that because the entire program must be stored in headers of a packet, the client cannot execute arbitrarily long programs, because the size of a program is constrained by the maximum packet size supported by the network. A potential workaround for this limitation is to require the client to chunk the program into multiple program pieces and stitch the program pieces together by setting the initial state for the next piece to the final state of the previous piece. This becomes non-trivial when we consider programs with branch instructions that reference instructions outside of

the current program piece. In that case, a potential solution is for clients to split their programs up into sufficiently small basic blocks whose return state indicates the next basic block to execute. The client is then responsible for shepherding the execution of the program by providing the system with the sequence of basic blocks as necessary. This solution suggests that it may be helpful to implement a client runtime that handles re-transmission of programs as well as chunking of programs. An alternative solution is to leverage the abstraction of "infinite" memory provided by the datastore to store the entire client's program within memory and incrementally execute the program via an interpreter "assembly" program that reads instructions from memory and executes them via interpretation.

4 Use-cases

A potentially realistic use-case for this distributed computing system is for network monitoring purposes. We can extend the "assembly" language with custom instructions that retrieve information about the state of the switch or the network.

As a proof of concept, this system could be used to implement a non-trivial program as a demonstration. Some examples include problems on Project Euler.