I am Xiangyu and I am a first-year phD student at NYU. Today, I will present our work in developing a code generator Chipmunk which automatically generate fast packet-processing code using program synthesis. This is joint work with Taegyun, Aatish and Anirudh from NYU and Srinivas from Rutgers.

The title of this talk is auto generating fast packet-processing code using programmable synthesis. So actually we mean generating these fast packet-processing code for programmable network substrates in this talk we especially focus on the programmable switch. And the main reason why people want to use programmable substrates to gradually replace the traditional fixed ones is that the former enjoy the function flexibility which is not shared by the later one. To be specific, in the background where various AI and IoT applications appear so quickly that people have more requirements for low-level network. However, traditional fixed network substrates have fixed functionality and users cannot fundamentally change it. If the designers want to include more functions into programmable substrates, it may take several years for this upgrade. In addition to programmability, with the growing link speeds, there is a need to speed up the running speed in programmable substrates by using faster packet-processing code. For example, SmartNICs run at line rates of 40-100 Gbit/s. Switches run at 100Gbit/s per port and a few Tbit/s in aggregate.

So with this emphasis on using programmable substrates and writing fast packet-processing code in them, let’s look at the difficulty to develop fast programs. There are two main reasons for this. First of all, developing such fast programs requires manual optimization by experts who are familiar with each underlying hardware architecture. For example, they must be aware of the cache and memory hierarchy for CPUs and SoC-based NICs; ALU, TCAM and SRAM limits for programmable switches; and of lookup tables, placement and routing for an FPGA. In reality, Microsoft hired a dedicated team of hardware engineers to program its FPGA-based SmartNIC. Then, you may ask the question: why no design an optimizing compiler to reduce the workload for manual optimization but do optimization automatically. This brings the second reason. I agree that an optimizing compiler will definitely alleviate the difficulties of generating such fast code, but building the optimizing compiler itself is arduous but fruitless because the building step requires significant engineering efforts, usually it will span decades, which is both time-consuming and money-consuming and cannot catch up with the evolving network hardware. Neither academic and industrial area have strong motivation to do this development.

So in response to the above concerns, here comes to the primary contribution to this talk: introducing program synthesis to develop code generator which we can use to generate fast packet-processing code. In other words, we map the code generation problem to one program synthesis problem and make full use of the existing program synthesis tools to solve the problem automatically.

Obviously, program synthesis problem face a key challenge: it will take quite a long time to solve it because it is a search problem over a large combinatorial search space of programs and the space grows exponentially with the number of bits in the hardware configurations. However, it is still meaningful to use it here mainly because of three reasons. First of all, most fast packet-processing programs are naturally small and simple (because they only need to support very easy algorithms in programmable devices), and if we use powerful machines and run our job in parallel, we can find the solution within reasonable time. Second, we currently there are more and more mature open-source synthesis tools and promising real-world applications of synthesis. We can foresee the appearance of new and well-developed synthesis tools which can speed up the problem largely. The third one is closely related to the real life situation.

Program synthesis is to automatically construct a program that satisfy given specification. Specification can be set of input/output examples (unit tests), natural language, first order logic expression or any other form that is easier to write the expected program. The early 21st century has seen a surge of practical interest in the idea of program synthesis in the [formal verification](https://en.wikipedia.org/wiki/Formal_verification) community and related fields. Since 2014 there has been a yearly program synthesis competition comparing different algorithms for program synthesis in a competitive event, the Syntax-Guided Synthesis Competition (or SyGuS-Comp). SyGus constrains the search space programs using syntactic restrictions.

The program synthesis tool we used for generating fast Pack-processing code is SKETCH, which originated from Prof. Armando’s phD thesis work.

The user will provide two programs: one is specification and the other one is partial program with holes in it. The goal is to find one kind of hole assignment which guarantee that the specification program and the partial program can be semantically equivalent within the input range we give. SKETCH here will complete either by filling in the hole values to make partial program become a full one if there is any feasible solutions or returning false if no possible hole assignment can satisfy semantic equivalence.

The concrete example is one spec program, we obtain the return value by multiplying the input value by 5. ??(2) means the one of the integer values choosing from 0 to 2^2-1which is 3. As for the first partial program, it is feasible because SKETCH can set ??(2) to be 2 while SKETCH cannot find the satisfying result for the second partial program.

Similar to the example shown here, the partial program with holes representing values within a finite range of integers, which can be used to encode the programmer’s insight into the structure of the implementation, ALU functionality in programmable switch to be specific. In our Chipmunk project, we will use developer’s program as the specification, use partial program to represent the structure of the substrate and use holes to represent large but finite number of low-level hardware configurations and then use program synthesis to test its feasibility.

This slide shows the operational mechanism by SKETCH. SKETCH use CEGIS algorithm to do program synthesis, here we call it internal cex mode for simplicity. **The program synthesis problem solves for hole values in the formula in first-order logic which is a QBF (quantified boolean formula) problem which is harder than SAT problem. Apart from that, there is no QBF solvers which are optimized for program synthesis. Therefore,** SKETCH uses CEGIS method to divide the whole program synthesis problem into two phases: synthesis phase and verification phase to reduce the computation complexity. The synthesis phase will use try to solve holes’ value assignments to satisfy the ‘partial’ inputs and the verification phase will verify the hole value assignments on all inputs which is specified by programmers. For example, if we set the input range to be 5 bits which is the default value of sketch. In the beginning, sketch will randomly choose subset of all inputs to do the synthesis problem, if it fails it means there should be no feasible solutions for this partial program; if it succeeds, it will continue to the verification stage to test more inputs. If we pass the verification, we are lucky enough to find the final solution otherwise, there is at least one input making the current hole assignment fail to work. So we add this counterexample into the input set and repeat synthesis again. Back and forth until we get the final result. Both synthesis and verification of CEGIS are simpler than directly solve the QBF problem because for each step we either fix test inputs or hole values which turns the QBF problem into a SAT problem which can be fed to a more efficient SAT solver.

Syntax-guided synthesis can be applied to code generation by using the developer’s program, say in C or P4, as a specification and using sketch to represent the structure of the substrate and using holes to represent a large but finite number of low-level hardware configurations such as assembly opcodes, operand choices for instructions and contents of look-up tables. All our benchmarks will be borrowed from benchmarks in Domino. Just as the program shown here, each program has both state variables and packet field. We will modify the stateful variables in Stateful ALU in Banzai simulator while change the value of packet field in stateless ALU. For our Chipmunk Banzai simulator, we will simulate the hardware architecture described in RMT and Banzai, commonly known as the Protocol Independent Switch Architecture (PISA) and the dominant architecture for high-speed programmable switches today, aiming at abstracting out a switch computation into a 2D grid of ALUs. The x axis represents pipeline stages (or we can call it depth) and the y axis represents parallel ALUs within a particular pipeline (or we can call it width). All packet fields values are stored in PHV (packet header vectors) and all ALUs’ computations are atomic so that the next packet arriving at that ALU a clock cycle later will see the updated value for stateful variables. For concrete ALU choice, we borrow from Domino that used to generate code for original program and develop stateless ALU based on Banzai’s stateless ALU which support arithmetic, boolean, relational and conditional operators. After providing the depth and width by user, Chipmunk will automatically generate a ‘partial program’ which mimic the operational mechanism inside the programmable switch and the rest of the work will be allocated for SKETCH to do programmable synthesis.

So now, hopefully I have convinced you that program synthesis can play an important role in freeing man power from generating fast packet-processing code. However, due to the complexity of synthesis problem, one major concern for this method is obviously the higher and more variable code generation time because program synthesis needs to solve QBF (here is SAT) problems whose total solving time increases exponentially as the number of holes increase. For example, if there are n holes, then the search space will be as large as 2^n and most of our benchmarks generate over 200 holes. Therefore, we come up with four main ideas listed above to speed up synthesis problem and I will discuss them one by one.

The first part is to reduce the constant values appearing in benchmarks. For example, the example shown in the slides is blue\_decrease.c in Domino program. We manually scan the program and downgrade all constants appearing in program into constants within 2-bit range. Here, we replace 10 by 2. The benefit of this transformation is that all constant/immediate holes appear in Chipmunk will only be 2-bit range which can dramatically reduce the total number of holes. Although we slightly change the semantics of some of benchmarks, but for these cases, we only focus on the compilation result so this change is reasonable. In future work, we will try to add constant synthesis algorithm to avoid semantic modification.

Previously, we introduce the CEGIS algorithm implemented in SKETCH. We call that internal counterexample mode program synthesis mode. We referred from that idea to develop our own version of CEGIS, we call it external counterexample mode. Instead of throwing the whole

synthesis problem with 10-bit input into sketch, we divide into two parts: synthesis & verification in 2-bit input; -> verification in 10-bit inputs -> if failed then add the counterexample in synthesis part so the synthesis step will be all 2-bit input + counterexample; else we are lucky enough to find a sol in smaller input range which satisfies 10-bit input.

In addition to modification in original program and our “own” external counterexample mode,we want to add more constraints which not only conserves the function of Chipmunk but also reduce the search space. In allocation case, we found that all possible position on the LHS can be mapped to one particular way in the RHS, so there must be some benefit if we choose to remove the freedom to force packet field to specific container.

For some benchmarks, there are only some of the simple operation in packet fields, so we can

choose simpler stateless ALU so that the total hole numbers can get slighted decreased. But there are still some disadvantages if we use simplified version of stateless ALU because it may convert from success to failure.

If permitted, we can make full use of the distributed system to run different stateless ALUs

In different machine and use optimal result. (fail from full stateless ALU or success from simplified stateless ALU).

For benchmarks, we started with eight programs drawn from multiple sources and these programs were previously compiled with Domino. Then, we mutated these programs in semantic-preserving ways to generate 10 mutations of each of the 8 programs. The way we generate mutations is based on the following ‘noise generation’ method, adding one obviously true condition in if statement, switch the if-else statement and switch the order of statement within if. The mutation generator will pass the program several times and add some ‘semantically equivalent modifications’ to the program. In theory, if Domino can generate code for original program successfully, it should do so for ten mutations. However, this is not the case.

Our current experiment results have shown the advantages of our Chipmunk mainly in two aspects. For completeness, although we can not guarantee that Chipmunk has 100% completeness for generating code but its performance has already much better than Domino.

Domino fails to generate code for most mutations of the original programs because it incorrectly concludes that the programs are too expressive to be implemented, which is avoided in Chipmunk because it reaches 100% for most of them which should be expected. After all, we can find a way to generate code for original programs. As for some programs which is rejected by Chipmunk are all because they reach the timeout we set for them (30 min per iteration).

If we provide enough time, they will finally provide the successful compilation result because we manually plug in hole values generate by other benchmarks and run verification on it, it generates the result.

Even for mutations both Domino and Chipmunk are successful, Chipmunk can generate code with fewer pipeline stages which is severely constrained in programmable switches (12 for Tofino). The reason that in some cases Chipmunk may consume more ALUs per stage is because sometimes, if we expand the pipeline to more stages, we can balance computation task to future stages and reduce the number of ALUs used per stage. But given the priority of number of pipeline stages, we prefer to use Chipmunk.

The initial experiment results look promising however we think this is still a beginning for this project. Throughout the whole process, we also develop many future directions worthwhile for further research. Here, I want to list the three most important ones for more open discussion. As for further speeding up, can we reduce the program synthesis time by replacing hot code regions with fast implementations using a database of localized code rewrite? As for realistic simulation, in addition to the program itself, how should we address other resource constraints like memory usage which depends on the workload? As for more usage in real programmable substrates, for example, can we generate code that runs on multi-core SmartNIC platforms?

To conclude, Chipmunk makes full use of program synthesis ideas to automatically generate fast packet-processing code which is more complete and hardware-resource saving at the cost of long compilation time. The current results encourage us to do more further research including speeding up synthesis time and implement this idea to more programmable substrates. I am welcoming for all feedback. Thank you.