VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

http://fit.vut.cz

Projektová praxe 2019/2020

Kompilátor jazyka P4.16 - Překlad kontrolního programu

Radek Veverka

Abstrakt
Cílem práce bylo seznámit se s překladačem jazyka P4.16 do VHDL, navrhnout, implementovat a otestovat jeho vybrané části. Tento překladač je vyvíjen v rámci projektu NFV společností CESNET. Hlavním úkolem bylo implementovat překlad P4 kontrolního programu v C++ tak, aby implementace byla architektonicky v souladu s ostatními částmi překladače a používala stejné návrhové vzory.

Klíčová slova: P4 — překladač — kontrolní program

Přiložené materiály: Oficiální repozitář překladače P4

xvever13@stud.fit.vutbr.cz, *Fakulta informačních technologií, Vysoké učení technické v Brně*

1. Úvod

Cílem projektu NFV200 je tvorba platformy, která umožní programování síťových aplikací a jejich nasazení na vysokorychlostní síťové karty. Karty obsahují konfigurovatelné hradlové pole FPGA, které je kompromisem mezi flexibilitou procesorů a rychlostí ASIC obvodů. Pro programování FPGA čipů se používá jazyk VHDL, který je syntetizačním nástrojem převeden do binární podoby. Binární kód je následně načten do hardwaru a FPGA čip se nakonfiguruje.

VHDL je ovšem jazyk poměrně nízkoúrovňový a popis složitějších aplikací v něm může být zbytečně pracný. Je proto výhodné použít jazyk, který poskytne vyšší úroveň abstrakce a usnadní programování. Jelikož středobodem síťových aplikací je zpracování paketů, nabízí se jazyk P4, který poskytuje veškerou potřebnou funkcionalitu.

2. Jazyk P4

Jazyk P4 existuje ve dvou základních specifikacích - starší P4.14 a novější P4.16. Práce se zabývá pouze překladem verze P4.16. P4 jazyk syntakticky vychází z jazyka C, nicméně sémanticky je naprosto odlišný a výrazně omezený. Je to z toho důvodu, že cílová síťová aplikace není primárně určena k sekvenčnímu zpracování na procesoru, jako je to u jazyka C, nýbrž má pevně danou strukturu, která odpovídá způsobu zpracování paketů. [2]

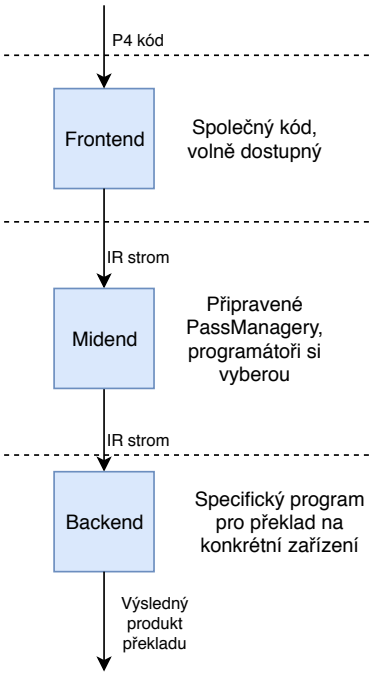
Základní struktura P4 programu je následující:

1. **Parsér** - Analyzuje obsah příchozího paketu a extrahuje jeho hlavičky do datových struktur jazyka P4 pro další použití. Funguje na bázi stavového automatu.
2. **Kontrolní program** - Rozhoduje o dalším zpracování extrahovaných hlaviček. Může aplikovat definované match+action tabulky, které na základě různých pravidel provádějí s hlavičkami akce (smazání, úprava hodnoty, atd.). Na základě výsledku aplikace tabulky nebo hodnot v hlavičkách kontrolní program rozhoduje, které další tabulky aplikuje.
3. **Deparsér** - Vybírá hlavičky z datových struktur P4 a skládá je do paketu k odeslání.

Jazyk P4 je navržen tak, aby nebyl závislý na konkrétním cílovém zařízení a aby bylo možné přidávat nové protokoly bez nutnosti kompletní výměny zařízení.

3. P4 překladač a jeho architektura

Jazyk P4 sám o sobě je pouze seznam lexikálních, gramatických a sémantických pravidel. Tato pravidla jsou podchycena v dokumentu specifikace jazyka. Pro praktické užití programovacího jazyka však potřebujeme překladač. Zde je velká výhoda P4, protože neexistuje žádný kompletní univerzální kompilátor.



Obrázek 1. Schéma architektury překladače P4.16

Každý výrobce zařízení podporující P4 jazyk si může naprogramovat vlastní kompilátor, který přeloží P4 na cokoliv, co jeho zařízení potřebuje. Není však nutné, aby každý programoval překladač kompletně, včetně lexikální a syntaktické analýzy, které jsou společné. Z toho důvodu je kompilátor rozdělen na 3 hlavní architektonické části:

1. **Frontend** - Společná část překladače, je k dispozici oficiální verze, pro P4.16 v C++, pro P4.14 v Pythonu. Vstupem je zdrojový kód P4, řeší lexikální, syntaktickou a části sémantické analýzy. Výstupem frontendu je abstraktní syntaktický strom, v C++ reprezentován jako množina propojených objektů s hierarchickou dědičností.
2. **Midend** - Úpravy stromu. Uživatel frontendu si může vybrat, které úpravy bude jeho překladač provádět. Jedná se zpravidla o optimalizace. Vstupem je abstraktní syntaktický strom, výstupem modifikovaný strom.
3. **Backend** - Samotný překladač, který prochází strom z midendu a tvoří cílový kód překladu. Jeho implementace je zcela na uživateli, nicméně oficiální repozitář překladače pro inspiraci obsahuje implementace různých backendů, například generátor obrázků s grafy abstraktního syntaktického stromu.

Aby bylo možné programovat vlastní backend, je nutné alespoň z části porozumět frontendu, především tak jeho výstupu. Základním stavebním kamenenem stromu produkovaného frontendem je tzv. IR uzel (Intermediate representation) [3]. Obecně lze říci, že tento uzel reprezentuje jednu entitu jazyka P4, například konstantu, operátor nebo control block. Každá konstrukce jazyka P4 má svůj typ IR uzlu ve frontendu, pričemž tyto typy uzlů od sebe různě dědí a vytvářejí tak hierarchii. Aby bylo možné abstrahovat a definovat společné operace nad všemi uzly, vrcholem v hierarchii dědičnosti je jedna společná třída zvaná IRNode.

Datová struktura přeloženého programu by byla samotná k ničemu, kdyby se nad ní nedaly provádět operace. Společná část překladače P4.16 nabízí několik způsobů, jak stromem procházet nebo jej měnit. Využit je návrhový vzor návštěvník (**visitor**). Ten odděluje logiku operace kompletně od stromu IR uzlů, což je zcela esenciální myšlenka, na které je kompilátor založen. Implementačně je návštěvník reprezentován třídou Visitor, přičemž existuje více typů návštěvníků, kteří od Visitora dědí, např. **Inspector**, **Modifier**, **Transform** nebo **PassManager**. Liší se především v oprávněních modifikace IR stromu, Inspector pouze může IR uzly číst, **Modifier** může měnit jejich obsah, **Transform** může přeskládávat celou strukturu stromu. Zajímavý a často používaný je **PassManager**, který po řadě spustí několik visitorů jako sekvenci. Několik PassManagerů obsahuje společná část kompilátoru - frontend (zde je nemůžeme měnit) a midend (můžeme nakonfigurovat, které se použijí).

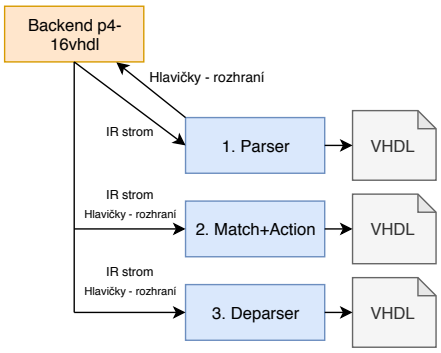
Implementace vlastního návštěvníka spočívá v tom, že nejprve oddědíme C++ třídu od typu návštěvníka, kterého chceme implementovat (např. Inspector) a poté v ní přepíšeme metody **preorder** nebo **postorder**. Tuto metodu můžeme nadefinovat pro kterýkoliv typ uzlu (konstantu, statement blok...). Metoda bere jako parametr zpracovávaný uzel, takže s ním můžeme provádět vše potřebné.

Po aplikaci návštěvníka na některý IR uzel stromu programu začne visitor "navštěvovat" všechny uzly podstromu. Při průchodu stromem volá **preorder** pro aktuální uzel, do kterého vstoupil a **postorder**, když se do uzlu vrací. Pokud jsme pro uzel v návštěvníkovi metodu přepsali, bude zavolána naše verze.

Společná část kompilátoru obsahuje také velké množství utilit, které mohou programátoři backendů využívat. Patří mezi ně například pěkně formátované chybové a varovné výpisy (s pozicí v P4 kódu), ladící výpisy nebo třeba konvertování P4 programu do JSONu a zpět.

4. p4-16vhdl backend

V případě projektu NFV200 je cílovým zařízením pro jazyk P4 hradlové pole FPGA, které je možné konfig-



Obrázek 2. Schéma architektury p4-16vhdl backendu

urovat kódem vysyntetizovaným z VHDL kódu. P4 program pro vysokorychlostní karty je tedy v podstatě překládán nadvakrát. Nejdříve se přeloží P4.16 kód pomocí kompilátor P4.16 do VHDL, který je následně přeložen patřičným nástrojem pro VHDL. Sémantické souvislosti mezi jazyky P4.16 a VHDL však nejsou jednoduché, protože jazyky se velmi liší, o této problematice pojednává disertační práce Pavla Benáčka [1], na které je způsob překladu založen. V této kapitole si nastíníme základní architekturu p416-vhdl backendu.

Kompilátor se skládá z několika C++ modulů, které jsou překládány samostatně a potom linkerem spojeny dohromady. Linker také zahrne společnou část kompilátoru, která je přeložena a linkována jako statická knihovna. Překladový systém je řízen programem CMake. Moduly jsou pak sdruženy do několika hlavních částí překladače, které do jisté míry kopírují již zmíněnou strukturu P4 programu.

Co se týče samotného generování VHDL kódu, **p4-16vhdl backend** se inspiruje známým architektonickým vzorem **MVC**, což znamená, že negeneruje kusy kódu přímo, ale vkládá je do připravených VHDL šablon. Výhodou je, že podstatná část generovaného kódu je oddělena od zdrojových kódů C++, což zajistí lepší rozšiřitelnost a jednodušší identifikaci chyb. Pro pohodlnou práci se šablonami je k backendu přilinkována šablonovací knihovna (**INJA**) a knihovna pro práci s JSON v C++ (**nlohmann**).

Před začátkem této práce byl hotový parsér, ale části s match+action tabulkami a kontrolním programem vyžadovaly zvýšenou pozornost. Byly už sice z velké části naimplementovány, ale bezohledně vůči návrhovému vzoru **visitor** a všeobecně v rozporu se základními praktikami objektově orientovaného programování. Toto vyústilo v situaci, kdy nebylo prakticky možné kompilátor dále rozšiřovat a chyby se ladily velice obtížně. Bylo jasné, že je kód třeba kompletně přepsat a ke každému modulu naimplementovat jednotkové testy, které zajistí, že základní stavební kameny překladače budou fungovat.

Kontrolní program, kterému se dále v práci věnuji, je blok mezi **parsérem** a **deparsérem**. Jde o hlavní řídící jednotku P4 aplikace. Na vstupu bere P4 kód a rozhraní, které v předchozím kroku vygeneroval parsér. Toto rozhraní se skládá ze signálů, které obsahují položky extrahované z hlaviček příchozích paketů. Data v hlavičkách zpravidla ovlivňují chod kontrolního programu, proto je důležité k nim mít v kontrolním programu přístup.

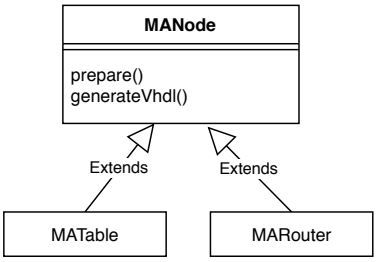
5. Překlad výrazů z P4 do VHDL

Překlad výrazů je na nejnižší úrovni a má nejméně závislostí, což je důvod, proč to byl můj první úkol. V tehdejší implementaci byly výrazy užity pouze na jednom místě, a to v podmínkách. Pro sémantické potřeby programů pro karty to zatím stačí, nicméně v budoucnu je možné, že působnost výrazů bude rozšířena. Vyplatí se tedy toto navrhnout a implementovat jako samostatný modul. Ve specifikaci jazyka P4 je spousta operací a typů operandů, které jsou podporovány ve výrazech, nicméně pro potřeby našeho překladače jsou využity pouze ty základní, které lze přímo přeložit do VHDL. Podporovanými operandy ve výrazech jsou pouze celočíselné kladné konstanty a položky extrahovaných hlaviček.

Hlavní myšlenka překladače výrazu je, že se pomocí **inspektorů** prochází strom z frontendu pro výraz. Během průchodu se inicializují třídy reprezentující části výrazu (**VhdlUnaryOp**, **VhdlBinaryOp**, **VhdlSingal**, **VhdlConstant**...). Všechny dědí od **VhdlNode**, tím pádem například binární operátory mohou mít dva odkazy na své operandy. Každý **VhdlNode** má implementovanou svoji metodu **getVhdl**(), která vrací část VHDL kódu. Tato metoda se zavolá na kořenovém uzlu výrazu a postupně se zanořuje, dokud nevrátí kompletní kód výrazu.

Rozdíl v P4 a VHDL výrazu spočívá především v zápisu operátoru. Například operátor **!=** je transformován na **/=**. Při překladu se zároveň kontroluje, jestli byl použit nějaký nepodporovaný operátor nebo operand, což způsobí výjimku. Toto byla také jedna z motivací kompletního přepisu této části překladače - špatný vstupní program od uživatele často způsoboval nedefinované chování.

Hlavím problémem překladu výrazů je určení, čím budou naplněny proměnné ve výrazech. Proto je součástí modulu pro výrazy také mapování signálů. Toto mapování může být dodáno buď uživatelem (vývojářem některé části překladače, která používá modul pro výrazy) nebo může být generováno automaticky. V případě automatického generování mapování umí modul vytvořit seznam potřebných VHDL signálů a VHDL přiřazení do nich.



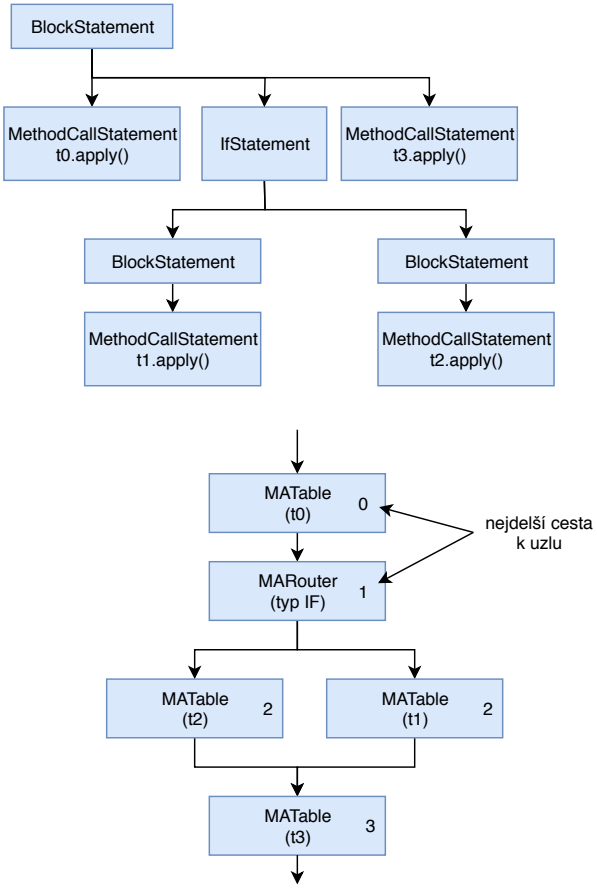
Obrázek 3. Schéma dědičnost match+action uzlů

6. Plánování match+action uzlů

Tento úkol je jádrem mé práce a kontrolního programu. Kontrolní program v P4 jazyce definuje, v jakém pořadí a za jakých okolností se budou **aplikovat tabulky**. Výstupem plánování je tzv. **match+action pipeline**, což je sekvence tabulek a směrovačů. Tyto entity jsou reprezentovány třídami **MATable** a **MARouter**, které dědí od společného předka **MANode**. Díky tomu je možné jejich uchování v jednom vektoru nehledě na typ. Tyto třídy jsou nejzásadnější nejen pro tvorbu match+action pipeline, ale pro celý překlad této části programu.

1. **MATable** je entita reprezentující tabulku. Je tvořena, když plánovač narazí v kontrolním programu na P4 volání **apply()**. Třída umí naplnit šablonu a vygenerovat kód pro analogickou komponentu ve VHDL. Tabulka není jednoduchá komponenta, proto je ještě rozdělena na podkomponenty - **search engine** a **action engine**. **MATable** neumí generovat kód pro akce a vyhledávání, ale díky objektovému návrhu je snadno napojena na další třídy překladače, které mají tyto podkomponenty na starost.
2. **MARouter** je entita reprezentující směrovač. Je tvořena, když plánovač narazí v kontrolním programu na větvení. To může být provedeno buď to vyhodnocením výrazu nebo na základě výsledku aplikované tabulky. **MARouter** provede naplnění VHDL šablony, která reprezentuje VHDL komponentu rozhodující o tom, který z následujících bloků v pipeline (směrovačů a tabulek) bude zpracován. Bloky v pipeline mezi aktuálním a zvoleným budou přeskočeny a nebudou mít na zpracování paketu žádný efekt. Směrovač používá modul pro překlad výrazů.

Většina tříd reprezentující komponenty (tabulky, action engine, search engine...) se drží podobného návrhu. Překlad probíhá ve dvou krocích - **prepare** a **generate**. Například po zavolání **prepare()** na **MATable** dojde k přípravě dat a naplnění šablony. Následně **MATable** zavolá **prepare()** na svém search enginu a



Obrázek 4. Nahoře schéma části abstraktního syntaktického stromu z P4 frontend, dole odpovídající nový strom s tabulkami a směrovači.

action enginu. Jedním voláním **prepare()** na komponentě na nejvyšší úrovni tedy dojde k zpracování všech potřebných částí. Operace **generate()** počítá s tím, že všechny části jsou již připraveny k samotnému překladu. Je generována adresářová struktura a výsledné soubory.

Problematickou částí plánovače však není samotná tvorba komponent, ale jejich řazení. Abstraktní syntaktický strom kontrolního programu z frontendu kompilátoru není vhodný pro určení pořadí, proto plánovač strom transformuje do grafu jiné podoby. Tento graf má jako uzly již zmiňované tabulky a směrovače. Každý uzel má navíc množinu svých následníků. Tabulky mají pouze jednoho následníka, neboť nevybírají z více uzlů ke zpracování. Směrovače jsou k výběru určeny a mají počet přímých následníku podle toho, o kterou rozhodovací konstrukci v kontrolním programu jde. Podmínka má dvě větve a tedy i dva následníky, konstrukce switch může mít různý počet větví a tedy i následníků. Na graf v této podobě lze aplikovat algoritmus pro nalezení nejdelší cesty, což je zároveň výsledná pipeline.

Jedním z hlavních úkolu překladače je hlásit chyby a nezpůsobovat nedefinované chování. Podobně jako u modulu pro výrazy i zde platí myšlenka, že co není povoleno, je zakázáno. Pokud tedy programátor v P4 použije konstrukci, která je sice specifikací jazyka povolena, ale nepodporována backendem pro VHDL, bude překlad ukončen s příslušným chybovým hlášením.

7. Propojení match+action uzlů

Plánovač na výstupu vyprodukuje pouze vektor match action uzlů, které najde v kontrolním programu. Tyto komponenty však nebudou samy od sebe fungovat bez řádného propojení. Nabízí se tedy vytvořit novou třídu **MATop**, která bude fungovat v podobném duchu jako ostatní komponenty překladače a bude mít za úkol generovat pipeline ve VHDL. Poslouží zároveň jako hlavní řídící jednotka match+action části kompilátoru. **MATop** získá na vstupu rozhraní signálů generované parserem, spustí plánovač a uloží si jeho výsledek. Rozhraní předá jednotlivým vygenerovaným uzlům.

Hlavním úkolem **MATop** je napojit výstupy z rozhraní jednoho uzlu do vstupů v rozhraní dalšího uzlu, který následuje v pipeline. Toto je ve VHDL nutné provést prostřednictvím pomocných signálů mezi entitami.

MATop má na starosti rovněž vygenerování těchto signálů. MATop je ještě nutné obalit další komponentou, která napojí její vstupy na výstupy parseru a výstupy na vstupy deparseru.

8. Testování

Součástí me práce bylo i psaní unit testů, které slouží k testování dílčích částí překladače. Zpravidla netestují integritu mezi moduly, nicméně výrazně ulehčují propojování částí dohromady.

P4 kompilátor používá knihovnu **GTest** pro C++ od Googlu. Tato knihovna obsahuje mimo jiné sadu maker, kterými lze testovat obsah hodnot proměnných. V adresářové struktuře backendu vznikla nová složka pro testy a byla zahrnuta do překladového systému. Pro každou část práce jsem vytvořil sadu unit testů. Ideálně by unit testy měly být na sobě nezávislé, prakticky je to ale téměř nemožné, protože komponenty jsou zpravidla závislé na jiných. Testy si pro překlad načítají jednoduché P4 programy.

Tento způsob testování je v podstatě jediný, který jsem kromě manuální analýzy výstupů při práci prováděl. Projekt ještě nebyl ve fázi, kdy by byly výsledky vysyntetizovatelné a spustitelné na kartách.

9. Závěr

Díky této práci jsem se blíže seznámil s jazykem C++ a VHDL. Zjistil jsem také, že vývoj aplikací nespočívá pouze v programování, ale v návrhu, testování a zisku informací. Poznal jsem, jak funguje překladač jazyka P4 a provedl reimplementaci jeho části a věřím, že to pomohlo P4 týmu se zaměřit na složitější aspekty projektu. V momentě, kdy jsem s prací končil, byl základ překladače téměř hotový a tým přecházel na ladění a syntézu VHDL kódu. Jelikož následující práce byly poměrně obtížné a pro mě nevhodné, začal jsem se orientovat ovladač karet programovaný v jazyce C.

Poděkování

Rád bych poděkoval Pavlu Benáčkovi, který mi zadával práci a vždy si našel čas, když jsem potřeboval s něčím pomoci. Mé díky patří také Tomáši Martínkovi, který mi poskytl informace k projektové praxi a zpětnou vazbu k technické zprávě.

Literatura

[1] BENÁČEK, P. *Generation of High-Speed Network Device from High-LevelDescription*. 2016. Disertační práce. Czech Technical University in Prague, Faculty of Information Technology. Dostupné z: https://fit.cvut.cz/sites/default/files/PhDThesis-Benacek.pdf.

[2] BENÁČEK, P. *Jazyk P4: budoucnost nastává*. Leden 2016. Dostupné z: https://www.root.cz/clanky/jazyk-p4-jako-budoucnost-sdn-dokonceni/.

[3] BUDIU, M. *P416 reference compiler implementation architecture* [URL: https://github.com/p4lang/p4c/blob/master/docs/compiler-design.pptx]. Červen 2018.