VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

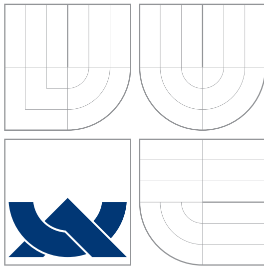
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

SOFTWARE PIPELINING V PŘEKLADAČI LLVM

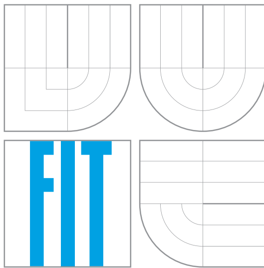
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE ONDREJ GLASNÁK
AUTHOR

BRNO 2014



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

SOFTWARE PIPELINING V PŘEKLADAČI LLVM

SOFTWARE PIPELINING IN THE LLVM COMPILER

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — ONDREJ GLASNÁK
AUTHOR

VEDOUCÍ PRÁCE — Ing. KAREL MASAŘÍK, Ph.D.
SUPERVISOR

BRNO 2014

// Abstrakt
Abstrakt

Tahle práce pojednává o návrhu a implementaci techniky programového zřetězení aneb Software Pipelining, optimalizaci cyklů v programu, která se snaží plně využít paralelismus na úrovni instrukcí. To dosahuje plánovaním instrukcí způsobem, aby se jednotlivé iterace cyklu překrývaly a bylo je možné vykonávat zřetězeně. Optimalizace takhle zvyšuje rychlost výsledného programu. Je tu popsaný návrh a implementace algoritmu Swing Modulo Scheduling, efektivní metody pro nacházení optimálního plánu pro zřetězení cyklů. Práce byla vytvořena jako součást většího projektu a to vývoje Codasip Framework. Jeho součástí je překladač jazyka C do jazyka symbolických instrukcí vytvořený nad překladačovou architekturou LLVM. V tomto překladači je implementován výsledek této práce.

Abstract

This thesis discusses a design and implementation of Software Pipelining, an optimization technique of loops in a program, which tries to exploit instruction-level parallelism. It is achieved by scheduling instructions in such a way to overlap iterations of the loop and therefore execute them in a pipeline. This way optimization speeds up the final program. There is a detailed description of design and implementation of Swing Modulo Scheduling algorithm, an effective and efficient method for finding near-optimal schedules for software-pipelined loops. This work has been done as a part of a larger project, the development of Codasip Framework. Part of this framework is the retargetable C compiler based on the compiler architecture LLVM, in which this work is implemented.

Klíčová slova

Modulo Scheduling, Softvérové zřetězení, Optimalizace cyklů, VLIW, Lissom, Codasip, LLVM, Plánování instrukcí

Keywords

Modulo Scheduling, Software Pipelining, Loop optimization, VLIW, Lissom, Codasip, LLVM, Instruction Scheduling

Citace

Ondrej Glasnák: Software pipelining v překladači LLVM, bakalářská práce, Brno, FIT VUT v Brně, 2014

Software pipelining v překladači LLVM

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Karla Masaříka Ph.D.

. . . .
Ondrej Glasnák
21. května 2014

Poděkování

Chcel by som poďakovať celému tímu Lissom za umožnenie spolupráce na ich projekte. Menovite ďakujem Ing. Adamovi Husárovi za rady a odbornú pomoc pri písaní práce. Svojej rodine, priateľom a priateľke ďakujem za ich podporu behom celého štúdia.

© Ondrej Glasnák, 2014.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Úvod **2**

2 Nástroje a architektúra **3**
2.1 Codasip Framework 3
2.2 LLVM 4
2.3 Architektúry 6

3 Plánovanie inštrukcií **8**
3.1 Typy plánovania 9
3.2 Software Pipelining 10
3.3 Modulo Scheduling 12
3.4 Swing Modulo Scheduling 17

4 Návrh a implementácia v LLVM **24**
4.1 Závislosť na architektúre 24
4.2 Umiestnenie v LLVM 25
4.3 Podpora LLVM 25
4.4 Implementácia 26
4.5 Výsledky 27

5 Možnosti ďalšej práce **30**

6 Záver **31**

A Návod na použitie a obsah CD **32**
A.1 Návod 32
A.2 Obsah CD 32

Kapitola 1

Úvod

Niektoré typy architektúr dovoľujú vykonávať viac inštrukcií paralelne, medzi nimi hlavne Superskalárne procesory, EPIC (Explicitly Parallel Instruction Computing) a VLIW (Very Long Instruction Word). Ďalšou charakteristikou VLIW a EPIC architektúr je, že výber inštrukcií, ktoré sa vykonajú a v akom poradí ponecháva na prekladači. Tento typ architektúry je tým pádom hardvérovo jednoduchší na úkor komplexnosti prekladača. Vyvstáva tu problém, ako vytvoriť optimálny plán inštrukcií a dosiahnuť čo najvyšší paralelizmus na úrovni inštrukcií. Vzhľadom na to, že sa jedná o NP-ťažký problém, v praxi sa používajú heuristické metódy, ktoré nachádzajú takmer optimálny plán, ale za zlomok času, ktorý by prekladač potreboval na nájdenie optimálneho plánu. Keďže väčšina času vykonávania programu je strávená v cykloch, existuje tu potenciál vykonávať viac inštrukcií naraz nájdením lepšieho plánu a vybrať, ktoré inštrukcie môžu byť vykonané paralelne s inými inštrukciami alebo v iný čas. Rieši sa to práve prelínaním iterácií. Táto práca pojednáva o návrhu a implementácii metódy pre nájdenie takého plánu pre telá cyklov. *Software Pipelining* je konkrétne technika, ktorá prelína jednotlivé iterácie cyklu a tým zvyšuje paralelizmus inštrukcií. Vykonáva inštrukcie z viacerých iterácií zároveň, čím sa rozširujú možnosti pre poradie vykonávania inštrukcií. Je to výhodné hlavne preto, že inštrukcie v jednej iterácii sú na sebe často závislé a táto technika otvára väčšie možnosti paralelizmu naprieč iteráciám. Najpoužívanejšiou triedou algoritmov pre dosahovanie *Software Pipelining* je tzv. *Modulo Scheduling.* Táto práca popisuje bližšie tieto metódy a ako dosahujú paralelizmus. Cieľom práce bolo vybrať algoritmus pre dosiahnutie *Software Pipelining*, navhnúť ho a implementovať v správnej fáze prekladača.

Práca bola vykonaná ako súčasť väčšieho projektu a to vývoja Codasip Framework. Jeho súčasťou je okrem iného prekladač jazyka C do jazyka symbolických inštrukcií vytvorený nad prekladačovou architektúrou LLVM. Keďže obecne pre dosahovanie reťazenia cyklov sú potrebné niektoré informácie o cieľovej architektúre, zaraďujú sa tieto algoritmy medzi tie závislé na architektúre. LLVM architektúra poskytuje tieto informácie pre širší okruh architektúr v a preto je algoritmus ľahko prenositeľný na iné architektúry.

Táto práca popisuje súčasný stav nástrojov a architektúry prekladača použitého na vývoj, ďalej sa venuje plánovaniu inštrukcií obecne a výberu algoritmu pre *Software Pipelining.* Popisuje detailne fungovanie rôznych metód z triedy *Modulo Scheduling.* Popisuje tiež návrh a detailné fungovanie algoritmu *Swing Modulo Scheduling*, jeho implementáciu v LLVM, testovanie, výsledky a celkový prínos tejto práce.

Kapitola 2

Nástroje a architektúra

Vzhľadom na fakt, že táto práca je súčasťou väčšieho projektu Codasip a je vyvíjaná nad architektúrou LLVM, je vhodné popísať tento projekt ako celok a význam, ktorý má práca z pohľadu tohoto projektu.

2.1 Codasip Framework

Pod projektom Codasip je vyvíjaný jazyk pre popis architektúr CodAL, odvodený od jazyku LISA (*Language for Instruction Set Architecture*). V tomto jazyku je možné popísať architektúru na dvoch úrovniach – na úrovni inštrukcií a na úrovni jednotlivých cyklov. Popis na úrovni inštrukcií umožňuje simuláciu, kde jedným krokom je inštrukcia, oproti tomu detailnejší popis na úrovni cyklov umožňuje simuláciu po jednotlivých cykloch procesora. Tento jazyk slúži pre jednoduchšie a rýchlejšie prototypovanie viacprocesorových systémov na čipe (Multiprocessor System-on-Chip, MPSoC) a aplikačne špecifických inštrukčne definovaných procesorov (*Application-Specific Instruction-set Processor*, ASIP).

Codasip Framework poskytuje nástroje a prostredie pre vývoj týchto procesorov pre konkrétne cieľové aplikácie. Umožňuje vytvoriť modely procesorov a z nich generovať nástroje pre programovanie a simuláciu. Medzi tieto nástroje patria assembler, disassembler, prekladač jazyka C, linker, simulátor, profiler, debugger a tiež je možné generovať popis architektúry v HDL jazyku. Táto práca je súčasťou jediného z týchto nástrojov a to prekladača jazyka C. Súčasťou tohoto Frameworku je aj Codasip Studio, ktoré poskytuje užívateľské prostredie založené na open-source vývojovej platforme Eclipse. Codasip Studio poskytuje rozšírenia tejto platformy na návrh procesorov v jazyku CodAL, vytváranie modelov, debugging a profilovanie ASIP a MPSoC.

2.1.1 Prekladač

Pre programovanie ASIP, Codasip Framework poskytuje okrem iného aj prenositeľný prekladač jazyka C, založený na open-source infraštruktúre LLVM, ktorá je podporovaná mnohými veľkými spoločnosťami, medzi nimi aj Apple a Google. LLVM (kedysi skratka pre *Low-Level Virtual Machine*) poskytuje mnohé optimalizácie a je v aktívnom vývoji. Codasip rozširuje LLVM hlavne o optimalizácie profilovania a podporu VLIW architektúr. Využíva profilovacie informácie a na ich základe vytvára väčšie bloky kódu, ktoré sa potom dajú plánovať. Takýmto plánovaním sa zvyšuje paralelizmus na úrovni inštrukcií

(*Instruction-Level Parallelism*, ILP) a tým zvyšuje efektivitu preloženého programu a využitie výhod VLIW architektúry.

2.1.2 Generovanie prekladača a preklad

Prekladač sa generuje viacerými krokmi, ako je zobrazené na obrázku 2.1. Po tom, ako je v Codasip Frameworku vytvorený model v jazyku CodAL, je možné ho preložiť cez CodAL prekladač do XML reprezentácie. Ďalším krokom je extraktor sémantiky. Túto časť je možné použiť len nad CodAL modelom, ktorý je popísaný na úrovni inštrukcií. Je založený na LLVM a skladá sa z viacerých priechodov, ktoré spracovajú vstupný model a rozlišujú jednotlivé inštrukcie. Inštrukcie, ktoré sa dajú použiť napr. nad dvoma registrami, ale aj nad registrom a priamym operandom, budú v sémantike obsiahnuté dvakrát. V extrahovanej sémantike budú teda všetky možné varianty použitia inštrukcií. Samotná sémantika je zapísaná najskôr v ANSI C a potom je preložená do LLVM IR. 2.2.2 Nad týmto medzikódom prebieha viacero optimalizácii a potom sa sémantika prevedie do špeciálneho tvaru, ktorý obsahuje informácie o jednotlivých inštrukciách ako syntax v jazyku assembler daného procesoru, binárne kódovanie, latenciu a podobne. Ak sa jedná o VLIW architektúru, nachádza sa tu tiež informácia o pozícii v *bundle*, balíku inštrukcií spojených do jednej inštrukcie pre paralelné vykonanie.

Táto extrahovaná sémantika je vstupom pre generátor backendu. Je to program, ktorý na základe tejto sémantiky vygeneruje backend podľa platformy LLVM. Ten sa spolu s LLVM knižnicami preloží do samostatného programu, ktorým je výsledný prekladač.

Samotný preklad prebieha nasledovne: Prekladaný program je nutné najprv preložiť do LLVM IR medzikódu pomocou frontendu, môžeme použiť napr. prekladač clang. Program v LLVM IR je možné preložiť pomocou generovaného prekladača do jazyku symbolických adries daného procesoru. Tento program pomocou nástroja assembler preložíme do binárneho objektového formátu a pomocou linkeru do binárneho formátu určeného pre daný procesor. Preložený program je možné odsimulovať použitím simulátora, z ktorého získame informácie o behu programu, teda výpis všetkých registrov, návratovú hodnotu programu, stav programu na konci simulácie, prípadné chyby, ktoré nastali pri simulácii, dĺžku simulácie, rýchlosť simulácie v MIPS (Million Instructions Per Second) a tiež počet procesorových cyklov.

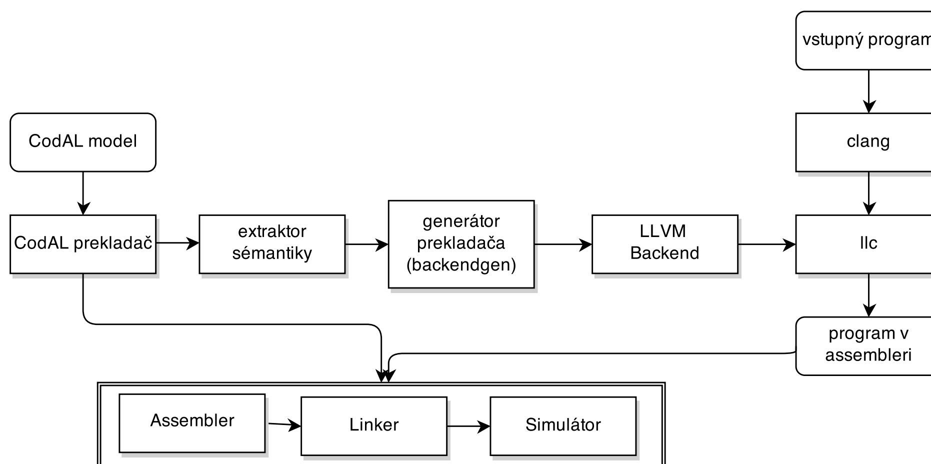
2.2 LLVM

Prekladačová platforma LLVM [18] zahŕňa kolekciu knižníc a nástrojov, ktoré umožňujú jednoduchší vývoj prekladačov, asemblerov, linkerov, JIT generátorov kódu a pod. LLVM je známe hlavne za *Clang*, prekladač C/C++/Objective-C, ktorý oproti GCC prekladaču ponúka množstvo vylepšení. Táto platforma je dnes bežne využívaná pre implementáciu mnohých kompilovaných jazykov, vrátane tých, ktoré sú podporované GCC, Java, .NET, Python, Ruby a mnohých iných menej známych jazykov. Taktiež nahradila niekoľko prekladačov, medzi inými tie používané v Apple OpenGL a Adobe After Effects.

Práve na tejto platforme je založené generovanie prekladačov, ktoré vyvíja Codasip.

2.2.1 Prekladač LLVM

Bežne sa prekladač delí na tri základné časti, a to Front-end, Optimalizátor a Back-end. Úlohou front-endu je analyzovať vstupný zdrojový kód a detekovať akékoľvek syntaktické



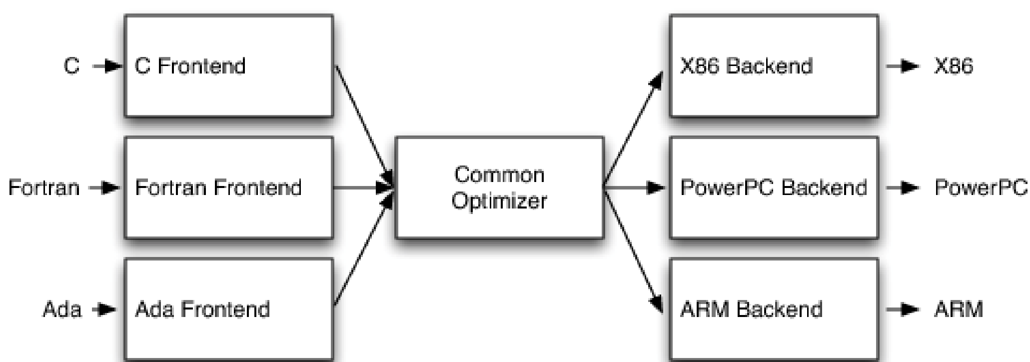
Obrázok 2.1: Generovanie prekladača [17]

či sémantické chyby v ňom. V tejto fáze sa tiež vytvára abstraktný syntaktický strom na reprezentáciu zdrojového kódu.

Optimalizátor ako stredná vrstva prekladača býva často nezávislá ako od architektúry, tak aj od jazyka, v ktorom je zdrojový kód zapísaný, a takto je to aj v LLVM. Optimalizátor vykonáva jednotlivé analyzačné a transformačné priechody nad medzikódom LLVM IR.

Nakoniec je tu back-end, ktorý generuje kód pre cieľovú architektúru. Je zodpovedný hlavne za generovanie správneho kódu, ktorý sa logicky nelíši od vstupného kódu. Ďalej je tu snaha využiť výhody a vlastnosti cieľovej architektúry pre lepšie generovanie kódu. Nachádza sa tu hlavne výber cieľových inštrukcií, posledné optimalizácie nad cieľovými inštrukciami a tiež alokácia registrov.

Výhodou takejto štruktúry je hlavne možnosť implementovať časť prekladača nezávislú ako od rôznych jazykov, tak od rôznych architektúr. Tým umožňuje väčšiu prenositeľnosť a rýchlejší vývoj prekladačov pre nové jazyky a architektúry. Naznačené na obrázku 2.2.



Obrázok 2.2: Prenositeľnosť LLVM prekladača [17]

2.2.2 LLVM IR

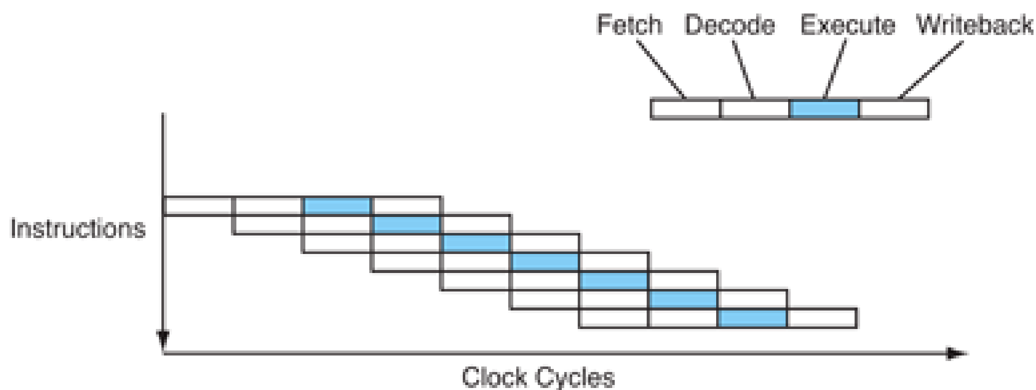
Dôležitou časťou LLVM platformy je jazyk LLVM IR (Intermediate Representation), jazyk na úrovni assembleru, avšak platformovo nezávislý. Nad týmto jazykom prebieha väčšina optimalizácií v LLVM. Je to medzikód v SSA (Static Single Assignment) forme, čo znamená, že do jednej premennej je možné priradiť hodnotu len jedenkrát v celom programe. Táto forma kódu zvyšuje možnosti optimalizácií dostupných v tejto fáze prekladu. Jazyk LLVM IR sa neskôr prevedie na inštrukcie cieľovej architektúry, nad ktorými prebiehajú ďalšie priechody v backende.

2.3 Architektúry

Existuje viacero architektúr, ktoré umožňujú svojou stavbou paralelizmus na úrovni inštrukcií. Líšia sa metódami, ktorými tento paralelizmus dosahujú. Vykonávanie jednej inštrukcie má niekoľko fáz, ktorými musí sekvenčne prejsť. Vykonanie v prípade RISC architektúry má fázy:

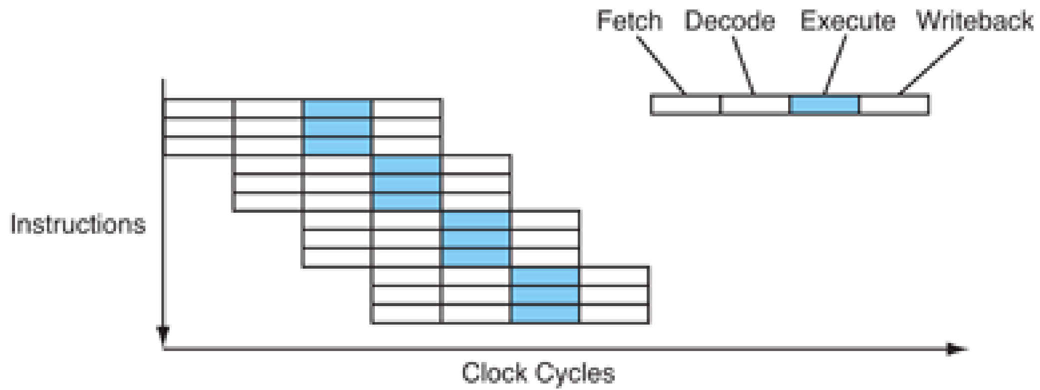
- Instruction fetch - vyzdvihnutie inštrukcie.
- Decode - dekódovanie inštrukcie, tu sa tiež načítavajú potrebné registry.
- Execute - vykonanie inštrukcie.
- Writeback - zápis výsledku do registrového poľa.

Aj nesuperskalárne jednoprocesorové mikroprocesory môžu spracovávať viacero inštrukcií naraz, tejto technike sa hovorí *pipelining*, zreťazené spracovanie inštrukcií. Bežné zreťazenie inštrukcií v RISC architektúre je znázornené na obrázku 2.3.



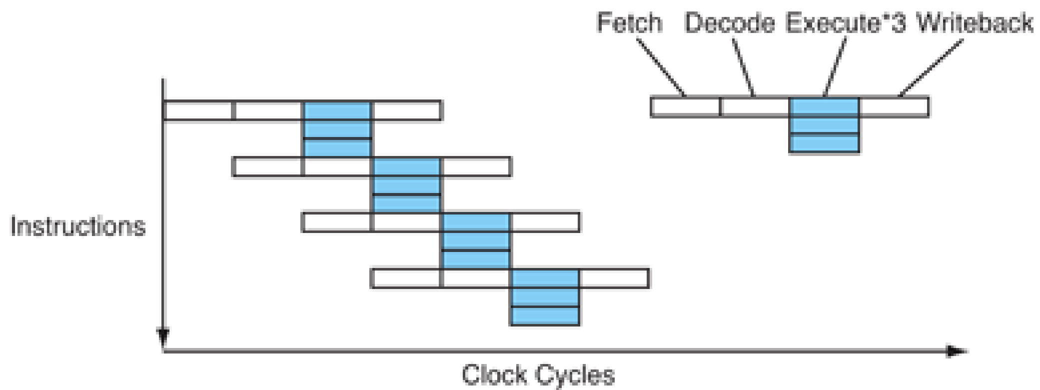
Obrázok 2.3: Zreťazenie inštrukcií na RISC architektúre

Superskalárne architektúry vznikli ako spôsob zvyšovania výkonu procesoru. Niektoré časti procesoru sú duplikované, napríklad matematický koprocesor (FPU) alebo aritmeticko-logická jednotka (ALU). Od viacjadrových procesorov sa líši tým, že nie sú duplikované celé jadrá, ale iba niektoré časti procesoru. Narastá tým komplexnosť architektúry, ale tiež počet inštrukcií vykonávaných za jeden cyklus procesoru. Zreťazenie je zobrazené na obrázku 2.4.



Obrázok 2.4: Zreťazenie inštrukcií na superskalárnej architektúre

VLIW sú podmnožinou superskalárnych architektúr a poskytujú niekoľko vylepšení. Príkladom je jednoduchší hardware, ktorého zložitosť je nahradená komplexnosťou prekladača. V jednej inštrukcii je zakódovaných viacero operácií, tie sú načítané ako jedna inštrukcia, spracované a vykonané paralelne na viacerých jednotkách, ako zobrazené na obrázku 2.5. Od superskalárnych architektúr sú jednoduchšie a tiež ponechávajú výber a poradie inštrukcií na prekladači. Skupina inštrukcií, ktoré sú združené do veľmi dlhého inštrukčného slova a vykonávané paralelne sa nazýva *bundle*.



Obrázok 2.5: Zreťazenie inštrukcií na VLIW

Kapitola 3

Plánovanie inštrukcií

Plánovanie inštrukcií má za úlohu vytvoriť poradie, v ktorom sa inštrukcie vykonávajú, a to tak, aby sa nezmenila logickú štruktúru vykonávaného kódu.

Tiež je žiadané predísť zbytočnému čakaniu medzi vykonávaním inštrukcií. Čakanie nastáva, keď sú inštrukcie na sebe dátovo závislé a výsledok z prvej inštrukcie je použitý ako operand inej. Vzniká niekoľko typov konfliktov, ktoré musí prekladač riešiť:

- konflikt typu RAW (Read-after-Write) alebo *true*, pravá závislosť nastáva, keď jedna inštrukcia načítava výsledok zapísaný inou inštrukciou. Inštrukcia, ktorá číta musí nasledovať tú, ktorá zapisuje, a to o určitý počet cyklov, aby bolo možné načítať správny výsledok bez čakania. Toto je najčastejší konflikt, ktorý nastáva pri reťazení na sebe závislých inštrukcií alebo vykonávaní inštrukcií v inom poradí, než zapísané v programe.

- konflikt typu WAR (Write-after-Read) alebo *anti* nastáva, keď inštrukcia zapisuje do operandu inej inštrukcie, ktorá je plánovaná pred ňou. Vzhľadom na to, že prvá inštrukcia potrebuje načítať operand pred prepísaním, musí sa zaistiť, aby tento operand nebol prepísaný skôr, než je vykonaná prvá inštrukcia. Opäť tu nastáva zdržanie o určitý počet cyklov, ktoré musia prebehnúť medzi týmito inštrukciami. Pri týchto typoch konfliktov však obvykle stačí, aby bolo zaistené, aby sa inštrukcia, ktorá načítava, vykonala pred inštrukciou, ktorá zapisuje. RAW konflikty medzi registrami obvykle existuje až po alokácii registrov. Na druhej strane tieto konflikty nastávajú pri prístupe do pamäte aj pred alokáciou registrov a aj pri zreťazenom sekvenčnom vykonávaní inštrukcií. Pre nás je dôležité, že v prípade superskalárnych procesorov a VLIW architektúr môžu byť inštrukcie s týmto konfliktom naplánované súčasne v jednom *bundle*.

- konflikt typu WAW (Write-after-Write) alebo *output* závislosť dvoch inštrukcií nastáva, keď sa obe snažia zapisovať na jedno miesto. Tým pádom by sa logicky výsledok zapísaný prvou inštrukciou nikde nevyužil, len bola prepísaná druhou (pri využití zapísanej hodnoty by to bola závislosť *true* nasledovaná závislosťou typu *anti*). V tomto prípade, ak nemá prvá inštrukcia iný vedľajší efekt než zápis tejto hodnoty, je zbytočná a môže sa odstrániť.

Z týchto vzájomných závislostí medzi inštrukciami prekladač tvorí graf závislostí, obdobu acyklického orientovaného grafu (DAG, *Directed Acyclic Graph*). Viacero metód využíva tento graf práve na získanie lepšieho plánu a informáciách o obmedzeniach a možnostiach plánovania.

Ďalším cieľom prekladača je zaistiť vyváženie rýchlosti prekladu, počet nutných registrov, ich životnosť a rýchlosť výsledného programu. Obvykle majú techniky na plánovanie za úlohu nájsť čo najkratší plán inštrukcií. Dĺžka plánu sa meria v počte cyklov, ktoré potrebuje procesor na vykonanie plánu.

3.1 Typy plánovania

Plánovanie inštrukcií sa delí podľa toho, v akom merítku je možné plánovať inštrukcie.

3.1.1 Lokálne plánovanie

Prebieha nad jedným základným blokom programu, časťou kódu, ktorá má jediný vstupný bod a jediný výstupný bod. Často sa používa varianta algoritmu *List Scheduling* s rôznymi heuristikami. Tento algoritmus začína vytvorením grafu závislostí medzi inštrukciami v základnom bloku. Postupuje po jednotlivých cykloch procesoru a v každom kroku identifikuje množinu pripravených inštrukcií, ktoré je možné naplánovať na konkrétny cyklus. Pripravenosť inštrukcie je zistená z grafu závislostí – nesmú tu vznikať dátové konflikty, a zároveň z obmedzení architektúry – počet dostupných funkčných jednotiek, na ktoré je možné inštrukcie plánovať. Výber z množiny pripravených inštrukcií je ponechaný na konkrétnej použitej heuristike. Výber je založený na prioritnej funkcii, ktorá uprednostňuje inštrukcie, ktoré majú potencionálny benefit pre lepší plán inštrukcií.

3.1.2 Globálne plánovanie

Ako názov napovedá, tento typ plánovania sa zameriava na väčšie úseky programu a usiluje o zvyšovanie paralelizmu na úrovni inštrukcií aj mimo základných blokov. Keďže lokálne plánovanie je limitované základným blokom (5 – 20 inštrukcií [29]), toto plánovanie zvyšuje potenciál pre lepší plán inštrukcií.

Prístupy k plánovaniu môžu byť rôzne, podľa [23] sa delia na metódy založené na profile (profile-driven methods) a na štruktúre (structure-driven methods). Metódy založené na štruktúre sa snažia nájsť paralelizmus redistribúciou inštrukcií naprieč všetkým možným cestám vykonávania programu. Patrí medzi ne hlavne *Percolation Scheduling* [25], ktorý sa skladá z štyroch transformácií kódu, ktorými sú *delete, move, move conditional* a *unify*. Pre každý uzol v grafe sa snaží aplikovať všetky štyri transformácie, kým je možné aplikovať aspoň jednu z nich. Ďalšie menej známe algoritmy sú *Trailblazing* [26] odvodený od *Percolation Scheduling* a *Meld Scheduling* [1].

Metódy založené na profile na druhú stranu využívajú profilovacie informácie na nájdenie frekventovane vykonávaných ciest v programe a vďaka týmto informáciám sú schopné nájsť lepší plán inštrukcií. Často vylepšia plán pre vykonávanie frekventovaných ciest na úkor tých menej využívaných. Ak sú však tieto profilovacie informácie korektné, finálny výkon je lepší vďaka zlepšeniu častejšie vykonávaných ciest.

Príkladom tejto metódy je *Trace Scheduling* [10], algoritmus, ktorý sa snaží vytvoriť cestu skrz program, ktorá sa skladá z najviac frekventovaných základných blokov. Táto cesta sa nazýva *trace*. Pre každý blok v *trace* platí, že má len jediného predchodcu aj následníka, získava tým vlastnosti základného bloku.

Superblock Scheduling [14] je odvodený od metódy *Trace Scheduling* a líši sa tým, že vytvára superbloky – bloky s jediným vstupom, môžu však mať viacero výstupov. Ako popísané v práci Ing. T. Mináča [24], *Superblock Scheduling* už bol implementovaný ako

súčasť prekladača v Codasip Frameworku. Využíva variantu *List Scheduling* popísanú vyššie.

Ďalšími príkladmi sú *Treeregion Scheduling* [4] zameraný na spájanie blokov do stromovej štruktúry a *Hyperblock Scheduling* [22], rozširujúci *Superblock Scheduling* o predikciu podmienok, čím sa kód linearizuje.

3.1.3 Cyklické plánovanie

Prebieha nad telom cyklu, teda viacerými základnými blokmi, ktoré majú cyklickú štruktúru. V najjednoduchšom prípade je to v podstate plánovanie nad základnými blokmi, ktoré sú jednotlivými iteráciami cyklu. Prvé metódy na dosiahnutie cyklického plánovania niekoľkokrát rozbalili cyklus a použili naň variantu globálneho plánovania.[11] Stále tu zostával nevyriešený problém, že na začiatku a konci jednotlivej iterácie boli funkcionálne jednotky málo využité. Bolo to tým, že väčšina cyklov má podobnú štruktúru - načítať dáta, aritmeticky upraviť tieto dáta a uložiť ich. Táto štruktúra má za následok menšiu priepustnosť, keďže tieto inštrukcie sú na sebe dátovo závislé, nedajú sa vykonať paralelne a musia čakať. Je teda žiadané začať vykonávanie ďalšej iterácie akonáhle je to možné a zvýšiť tým priepustnosť a využitie funkcionálnych jednotiek procesoru. Dnes sa pod cyklickým plánovaním rozumie hlavne *Software Pipelining* [3], prelínanie vykonávania inštrukcií z viacerých iterácií cyklu. Má za cieľ udržať vysoké využitie pipeline naprieč iteráciám.

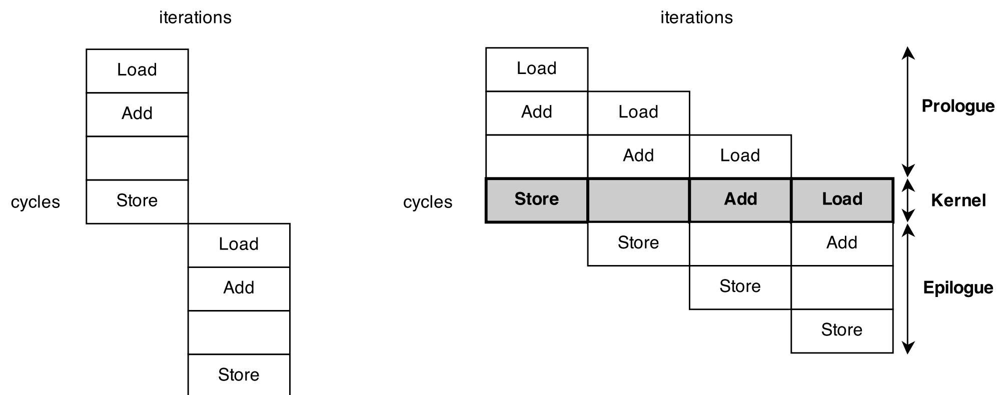
3.2 Software Pipelining

Ako bolo spomenuté vyššie, *Software Pipelining* je forma cyklického plánovania, teda zameriava sa na prelínanie iterácií cyklov. Koncept je znázornený na obrázku 3.1. Vľavo je vidieť sekvenčné vykonávanie jednoduchého cyklu pozostávajúceho z troch inštrukcií *Load, Add, Store* v dvoch iteráciách. Vpravo sú zobrazené štyri iterácie v poradí, v akom sú vykonávané s použitou technikou *Software Pipelining*. Jednotlivé iterácie sú inicializované v intervale II (*Initiation Interval*). II sa počíta v cykloch procesoru a v príklade na obrázku platí $II = 1$. Keďže v jadre cyklu (kernel) vykonávame niekoľko iterácií súčasne a pre inštrukcie musia byť vyriešené dátové závislosti, rozdeľuje sa cyklus na tri časti: prológ, kernel a epilóg. Prológ je vykonávaný iba raz a slúži na vykonanie potrebných inštrukcií predtým, ako môžeme začať vykonávať kernel. Kernel v tomto prípade prelína 4 iterácie a môže byť vykonaný až po skončení prológu, v čase vykonania i-tej iterácie, kde i je počet iterácií, ktoré kernel prelína (v tomto prípade $i = 4$). Kernel sa vykonáva cyklicky $N - i + 1$ krát, kde N je celkový počet iterácií.[1] Cyklus končí epilógom, ktorý slúži na dokončenie iterácií, ktoré začnú svoje vykonávanie v kerneli. Táto časť sa opäť vykonáva iba raz.

Táto metóda je využiteľná aj pre RISC architektúry, najväčší benefit však má pre VLIW architektúry, ktoré dokážu vykonávať viacero inštrukcií naraz. V príklade 3.1 sa taktiež počíta s VLIW architektúrou s aspoň štyrmi funkcionálnymi jednotkami.

Existujú architektúry, ktoré majú hardvérovú podporu tejto optimalizácie. Príkladom je architektúra Intel IA-64 [5] a predtým Cydra 5 [8]. Táto podpora spočíva v pridanom rotujúcom súbore registrov, ktorý dovoľuje prelínať iterácie a striedať používané registre rôznymi iteráciami. Väčšina architektúr však dnes v tomto spolieha na prekladač.

[1] Pri použití *Modulo Variable Expansion* to nemusí platiť, keďže sa rozbaluje kernel a vykonáva viac než jednu iteráciu v intervale II.



Obrázok 3.1: Vykonávanie iterácií sekvenčne (vľavo) a so *Software Pipelining* (vpravo)

Nájsť optimálny plán pre prelínanie iterácií na reálnej archietktúre s obmedzeným počtom funkcionálnych jednotiek je NP-úplný problém. [16] Problém nájdenia plánu sa dá totiž previesť na NP-úplný problém lineárneho programovania (optimalizácie) nad celými číslami (*Integer Linear Programming*). Boli porovnané optimálne a heuristické prístupy k nájdeniu čo najlepšieho konečného plánu pre *Software Pipelining* a bolo zistené, že heurstické metódy dokážu nájsť takmer optimálny plán za zlomok času potrebného na nájdenie optimálneho plánu. [28] Odvtedy bolo navrhnutých množstvo metód a heuristík, ktoré dosahujú rôzne plány, niektoré aj také, čo dosahujú optimálny alebo takmer optimálny plán vo väčšine prípadov.

Rau [27] delí algoritmy na dosiahnutie *Software Pipelining* a dve základné triedy, a to *move-then-schedule* a *schedule-then-move*. Techniky *move-then-schedule* premiestňujú jednu po druhej inštrukcie do nasledujúcich alebo predošlých iterácií a potom ich plánuje, aby dosiahol zreťazený plán inštrukcií. Tým sú v jednom bloku vykonávané viaceré iterácie zároveň. Tieto transformácie nezasahujú do štruktúry cyklu, ale vytvárajú nové poradie inštrukcií, teda tvorí nové telo cyklu. Tým sa vyrieši obvyklá závislosť inštrukcií v jednotlivých iteráciách a algoritmus dosahuje vyššieho paralelizmu.

Na druhej strane sú techniky *schedule-then-move*, ako názov napovedá, najprv vytvárajú konkrétny plán inštrukcií a až potom hýbe inštrukciami. Existujú dve metódy, ako toto dosiahnuť.

Prvou je tzv. *Unroll-while-Scheduling*, niekedy tiež *Kernel Recognition* postupne rozbaluje a plánuje cyklus až dokým sa nedostane do stavu, kedy by bol zvyšok plánu len opakovaním existujúceho kódu. Tento algoritmus však môže spôsobovať extrémnu expanziu kódu a potrebného času na optimalizáciu, pretože na nájdenie takého opakovania plánu medzi iteráciami môže byť nutné rozbaliť cyklus niekoľko sto krát. Preto sa väčšina výskumu v tomto smere zameriava na iné metódy pre dosahovanie *Software Pipelining*.

Druhou metódou *schedule-then-move* je prístup menom *Modulo Scheduling*. [6] Má za cieľ vytvoriť plán inštrukcií pre jednu iteráciu, a to taký, že ak budeme tento plán opakovať, všetky závislosti medzi inštrukciami sú zachované a nevznikajú tu žiadne konflikty v iterácii ani naprieč iteráciám. Špecifikuje množinu pravidiel pre vytvorenie správneho plánu, ktoré musia byť dodržané počas vytvárania tohoto plánu. Je to relatívne jednoduchý a efektívny prístup a bolo navrhnutých množstvo techník, ako takýto plán nájsť. Empiricky bola týmto

metódam dokázaná zložitosť $O(n^2)$, kde n je počet inštrukcií v tele cyklu. [29] Tiež bolo dokázané, že vo väčšine prípadov nachádza takmer-optimálne až optimálne plány. [28] Pre tieto dôvody bola tiež varianta *Modulo Scheduling* metódy zvolená pre cieľ tejto práce.

3.3 Modulo Scheduling

Modulo Scheduling je v podstate variantou vyššie spomenutého algoritmu *List Scheduling* pre cykly. 3.1.1 Inštrukcie sú zoradené do prioritného zoznamu a potom je z nich postupne vytváraný plán jednej iterácie.

Prvým krokom algoritmu je vytvorenie grafu závislostí (*Data Dependency Graph*, ďalej len DDG). V klasickom DAG (*Directed Acyclic Graph*) bežne využívanom na plánovanie sa nevyskytujú cykly, keďže dve inštrukcie nemôžu byť zároveň dátovo závislé, dochádzalo by k ich vzájomnému zablokovaniu. V jednotlivých iteráciách však môžu nastať tzv. rekurencie, ak je inštrukcia dátovo závislá od inštrukcie z jednej z predošlých iterácií. Hrany v takomto DDG majú vlastnosť *iteračnej vzdialenosti* d, ktorá označuje počet iterácií medzi závislými inštrukciami. Týmto vznikajú cykly v grafe, s ktorými musí algoritmus pre *Modulo Scheduling* pracovať.

Algoritmus *List Scheduling* však pracuje nad acyklickým grafom a nemožno ho jednoducho použiť na cyklický graf. Lam [16] k tomuto problému pristupuje hierarchicky. Z rekurencií vytvára silno prepojené komponenty (*Strongly Connected Components*, SCC) v grafe a plánuje inštrukcie v tomto podgrafe. Plán tohoto podgrafu jednej SCC sa ďalej berie ako jedna makro-inštrukcia. Tým sa graf stáva acyklickým a je možné použiť variantu algoritmu *List Scheduling*. Tento hierarchický prístup tiež dovoľuje plánovanie cyklov, ktoré obsahujú vnorené cykly alebo podmienené príkazy.

Je nutné zobrať do úvahy, že *Software Pipelining* môže zvýšiť počet potrebných registrov v novom pláne inštrukcií. Pri prelínaní iterácií vzniká stav, kedy sú potrebné registre ešte z predošlých iterácií. Preto sekundárnym cieľom mnohých metód je aj redukovanie potrebných registrov. Ak je totiž potrebných viac registrov, než architektúra poskytuje, registre naviac sa uložia do pamäte a sú načítané znova, keď sú potrebné. Tento stav nie je žiadaný, keďže čítanie a zapisovanie do pamäte je v porovnaní s prácou s registrami časovo náročnejšie. Okrem toho predstavuje tento stav tiež problém pre alokáciu registrov. Životnosť jednej premennej v cykli sa prekrýva, keďže je potrebných viac verzií tejto hodnoty z viacerých iterácií. Prekrývanie životnosti by znamenalo prepísanie starej hodnoty za novú, avšak v tomto prípade je stará hodnota stále potrebná.

Jedným z navrhnutých riešení je *Modulo Variable Expansion*, ktorý rozbaluje kernel k-krát a strieda používané registre, aby bola ich životnosť maximálne $k * II$ cyklov. [16] Tým dostanú prelínané životnosti rovnakých premenných rôzne alokované registre a nie sú prepísané novými hodnotami kým sú potrebné. Je však nutné, aby mala architektúra dostatočný počet registrov pre rozbalený kernel, konkrétne k-násobok. Alternatívou je už spomenutá hardvérová podpora napr. IA-64 a Cydra architektúr.

Základom algoritmu je nájdenie MII (*Minimum Initiation Interval*), minimálneho počtu cyklov procesoru, v ktorom je možné inicializovať jednotlivé iterácie za sebou. Je to vlastne minimálny počet cyklov, ktoré musia prejsť medzi dvoma rovnakými inštrukciami z dvoch po sebe idúcich iterácií. Jedná sa o obrátenú hodnotu priepustnosti, takže pre zvýšenie priepustnosti je dôležité, aby bol tento interval čo najmenší.

Modulo Scheduling prístup definuje podmienky, ktoré musia byť dodržané aby vznikol správny finálny plán.

- **Vlastnosť Modulo**: Dve inštancie rovnakej inštrukcie z dvoch po sebe idúcich iterácií musia byť od seba vzdialené *II* cyklov. Teda ak je inštrukcia a z iterácie i naplánovaná na čas t, potom ďalšia inštancia tejto inštrukcie je naplánovaná na čas $t + II$.

$$t(a_i) + II = t(a_{i+1})$$

- **Dátové závislosti**: Pre každú závislosť $a_i \to b_j$, inštrukcia a_i musí byť naplánovaná aspoň $(i - j) * II$ cyklov pred inštrukciou b_j.

$$t(a_i) + (j - i) * II \leq t(b_j)$$

- **Obmedzenie zdrojov**: V žiadnom kroku vykonávania nemôže byť žiadna hardvérová jednotka alokovaná viac než jednou inštrukciou.

Pre jednoduchosť je používaný čas naplánovania prvej inštrukcie z prvej iterácie nulový, teda $t(a_0) = 0$.

3.3.1 Výpočet MII

Pre dodržanie podmienok je nutné vypočítať *MII.* Táto hodnota je vypočítaná z dvoch hodnôt, ktoré vyplývajú z podmienok pre *Modulo Scheduling.* Z dátových závislostí sa počíta hodnota RecMII (*Recurrence Minimum Initiation Interval*) a z obmedzení zdrojov sa vypočíta ResMII(*Resource Minimum Initiation Interval*).

- **RecMII**: Pre výpočet tejto hodnoty musíme nájsť všetky rekurencie v grafe. Pre každú rekurenciu R je vypočítaná celková hodnota *II* ako súčet latencií všetkých inštrukcií L a celkovej iteračnej vzdialenosti D. Výpočet $RecMII$ je potom spočítaný ako ich maximum.

$$RecMII = \max_{\forall R} \left\lceil \frac{L_R}{D_R} \right\rceil$$

- **ResMII**: Pre dodržanie podmienky obmedzených zdrojov je nutné vypočítať aj túto hodnotu. Opäť je to hodnota spočítaná ako maximum jednotlivých hodnôt, v tomto prípade funkcionálnych jednotiek na vykonávanie inštrukcií. Zoberme do úvahy všetky typy funkcionálnych jednotiek r, počet jednotlivých jednotiek, na ktoré sa dajú mapovať inštrukcie F_r a nakoniec celkový počet inštrukcií, ktoré sa na tomto type jednotiek vykonáva N_r. Inštrukcia zaberá túto jednotku na i cyklov. Potom platí, že ResMII je maximum týchto hodnôt:

$$ResMII = \max_{\forall r} \left\lceil \frac{N_r * i}{F_r} \right\rceil$$

Hodnota MII je nakoniec vypočítaná ako

$$MII = \max(RecMII, ResMII)$$

Metódy *Modulo Scheduling* sú obvykle rozšírením algoritmu *List Scheduling.* Líšia sa hlavne v tom, ako vyberajú inštrukcie do prioritnej fronty a ako konštruujú finálny plán. Napriek rôznym heuristikám sa snažia ako prvé plánovať inštrukcie, ktoré sú podľa nich náročné na plánovanie. Väčšinou sú to inštrukcie v rekurencii, špecificky v kritickej rekurencii s najvyššiou hodnotou II. Typicky algoritmus začína vytvorením DDG

a následným výpočtom MII. Aby sa vyhol konfliktom obsadenia zdrojov, vytvára *Modulo Reservation Table* (*MRT*), v ktorom je uložená informácia o obsadení hardvérových jednotiek v jednotlivých cykloch kernelu. Je to tabuľka $R \times II$, kde R je počet funkcionálnych jednotiek a II je dĺžka kernelu v cykloch. Kvôli vlastnosti Modulo ani nemôžu byť dve inštrukcie naplánované na jeden slot v *MRT*.

3.3.2 Iterative Modulo Scheduling

Iterative Modulo Scheduling (IMS) [27] využíva jednoduché rozšírenie nad *List Scheduling*. Začína tým, že vybuduje bežný graf závislostí s pridanými uzlami *START* a *STOP*. Uzol *START* je predchodcom všetkých ostatných uzlov v grafe a uzol *STOP* je ich následníkom. Využíva prioritnú frontu na základe výšky v grafe, ktorá je počítaná ako vzdialenosť od uzlu *START*.

Algoritmus pokračuje vyššie popísaným výpočtom *MII*. Túto hodnotu používa ako začiatočné *II*, podľa ktorého sa snaží nájsť finálny plán. Od algoritmu *List Scheduling* sa líši niekoľkými vlasnosťami.

Namiesto toho, aby plánoval všetky inštrukcie, ktoré je možné naplánovať v konkrétnom čase, hľadá optimálny čas, kedy plánovať jednotlivé inštrukcie. Tiež môžu byť inštrukcie plánované viackrát. IMS tiež definuje hodnotu *estart*, ktorá určuje najskorší čas, kedy sa môže inštrukcia naplánovať. Pretože inštrukcie môžu byť preplánované a tým sa hodnota *estart* môže meniť, IMS si priebežne ukladá hodnoty minulých *estart* pre jednotlivé inštrukcie a používa buď súčasnú hodnotu (ak je menšia než predošlá hodnota *estart*) alebo hodnotu $estart + 1$ z predošlých plánov. Toto zaručuje, že sa IMS nezasekne pri jednej inštrukcii a neplánuje ju znovu a znovu.

Algoritmus má dopredu definovanú maximálnu hodnotu II, pre ktorú hľadá finálny plán. Ak nenájde dostačujúci plán pre maximálnu hodnotu II, končí a originálny cyklus zostane neporušený. Keď IMS nájde poradie, v ktorom plánovať, určí časové intervaly, kedy je možné jednotlivé inštrukcie naplánovať podľa ich predchodcov, ktoré sú už vo finálnom pláne. Vypočíta *estart* podľa bezprostredných predchodcov, aby sa zachovali závislosti. Závislosti následníkov sú zachované tým, že ak by mal nastať konflikt medzi plánovanou inštrukciou a jej následníkom, nasledník je odstránený z plánu a vložený späť do prioritnej fronty.

Bola vykonaná rozsiahla štúdia zaoberajúca sa efektívnosťou IMS a iných metód pre *Modulo Scheduling*. Ukázalo sa, že IMS ako technika dosahuje takmer optimálny finálny plán vo väčšine prípadov, ale potrebuje na to výrazne viac registrov než iné testované metódy.

3.3.3 Slack Modulo Scheduling

Slack Modulo Scheduling [13] (ďalej len Slack) je ďalší algoritmus podobný bežnému *List Scheduling*. Narozdiel od IMS má namiesto výšky v grafe poradie vo fronte podľa tzv. „Slack“, teda stupňa voľnosti, ktorú má inštrukcia v tom, kde môže byť naplánovaná. Ďalším rozdielom je to, že táto stratégia plánuje obojsmerne, niektoré inštrukcie čo najskôr a niektoré čo najneskôr. Toto má za následok lepší plán, ale aj dlhšiu dobu kompilácie. Berie do úvahy dátové závislosti, obmedzené množstvo prostriedkov, počet dostupných registrov a aj kritické cesty, aby limitovala spätné trasovanie (backtracking).

Algoritmus opäť začína postavením grafu závislostí. Z neho a z informácií o cieľovej architektúre vypočíta MII. Aj Slack využíva dve pseudo-inštrukcie, `start` a `stop`. Rovnako ako u IMS, *start* je predchodcom všetkých ostatných uzlov v grafe, analogicky je *stop* následníkom všetkých uzlov. Inštrukcia *start* je naplánovaná na úplný začiatok, na cyklus 0,

stop sa plánuje tak ako všetky ostatné inštrukcie. Tieto uzly s pseudo-inštrukciami zaisťujú, že všetky inštrukcie budú mať aspoň jedného následníka a predchodcu.

Slack tiež určuje pre každú inštrukciu hodnoty *estart*(*Earliest Start*) a *lstart*(*Latest Start*). Veľkosť tohoto intervalu je „Slack“, ktorý algoritmus používa pre prioritnú frontu. Tiež určuje premennú *minDist*, ktorá určuje minimálny počet cyklov medzi dvoma inštrukciami. Ak nie je žiadna závislosť medzi inštrukciami, *minDist* je nastavená na hodnotu $-\infty$. Tiež je dôležité spočítať všetky *minDist* hodnoty pre každé nové *II*.

Algoritmus teda spočíta tieto prvotné hodnoty a potom vyberá inštrukciu, ktorá sa má plánovať. Vyberá podľa minimálnej hodnoty *Slack* a sekundárne podľa nižšej hodnoty *lstart*.

Ďalej algoritmus vyberá cyklus, v ktorom naplánuje vybranú inštrukciu. Sofistikovaná heuristika analyzuje tok inštrukcií a rozhodne, či inštrukciu naplánovať čo najneskôr alebo čo najskôr. Ak by na tomto mieste nastal konflikt naplánovaním tejto inštrukcie, budú konfliktné inštrukcie odstránené z plánu a pridané späť do prioritnej fronty. Opäť je tu mechanizmus, ktorý zabraňuje opakovanému preplánovaniu jednej inštrukcie donekonečna.

Po naplánovaní vybranej inštrukcie sú nanovo spočítané hodnoty *estart* a *lstart*. Pokračuje sa výberom ďalšej inštrukcie a jej plánovaním.

Ak sú inštrukcie odstraňované z plánu príliš často, celý plán je odstránený a inkrementuje sa hodnota *II*. Potom sa algoritmus opakuje dokiaľ nedosiahne finálny plán. Ak sa to nepodarí, zvyšuje *II* postupne na maximálnu hodnotu a ukončí sa.

Podľa už spomenutej komparatívnej štúdie, ktorá porovnala jednotlivé metódy Modulo Scheduling, Slack na väčšine architektúr našiel takmer optimálny plán a využil o dosť menej registrov ako IMS metóda. Tento fakt sa odrazil na vyššom čase kompilácie. IMS metóda nachádzala lepšie plány pri komplexnejších architektúrach. SMS oproti tomu väčšinou dokáže nájsť lepší plán než Slack na jednoduchších a stredne komplexných architektúrach, s podobným využitím registrov a v lepšom kompilačnom čase.

3.3.4 Hypernode Reduction Modulo Scheduling

Hypernode Reduction Modulo Scheduling (HRMS) [21] je ďalšia z heuristických techník triedy Modulo Scheduling. Dosahuje takmer optimálny plán. Táto technika sa zameriava na skrátenie životnosti inštrukcií, aby limitovala potrebu pre prílišné množstvo registrov. Názov *Hypernode Reduction* popisuje algoritmus pre zoraďovanie inštrukcií pre plánovanie, kedy sú všetky uzly v grafe postupne redukované na jeden alebo viac *Hyperuzlov*. *Hyperuzol* je vlastne uzol, ktorý popisuje buď samostatný uzol v grafe alebo celý podgraf DDG. Ak sú všetky inštrukcie navzájom závislé a teda graf je súvislý, vzniká z grafu postupnou redukciou jeden *Hyperuzol*.

Ako ostatné techniky, algoritmus začína postavením DDG a vypočítaním MII. *HRMS* je špecifické tým, ako zoraďuje inštrukcie do fronty na plánovanie. Fáza zoraďovania inštrukcií zaručuje, že inštrukcia bude mať iba predchodcov a následníkov v tomto čiastočnom pláne, jedinou výnimkou sú rekurencie, ktoré majú prioritu. Toto zoraďovanie je vykonané iba raz, takže aj keď sa zvyšuje *II*, poradie inštrukcií zostáva rovnaké. Je vykonané ako iteratívny algoritmus a pre každú iteráciu tohoto algoritmu nájde všetkých následníkov a predchodcov Hyperuzlu, zoradí tieto uzly a zlúči ich do jedného Hyperuzlu. Algoritmus pokračuje, kým je možné robiť tieto redukcie.

Pre DDG bez rekurencií prebieha algoritmus takto:

1. Výber začiatočného Hyperuzlu. Môže ním byť ktorýkoľvek uzol v DDG, často sa vyberá prvý uzol.

2. Nájsť všetky susedné uzly Hyperuzlu v grafe.
3. Nájsť všetky cesty medzi následníkmi a predchodcami Hyperuzlu.
4. Tieto uzly sú označené ako podgraf a redukované do jedného Hyperuzlu.
5. Uzly v tomto podgrafe sa topologicky zoradia a tento zoznam uzlov/inštrukcií je pridaný do konečnej fronty.
6. Tieto kroky sú opakované až dokým nie je možné vykonať žiadnu redukciu.

V prípade existencie rekurencií sa nevyberá začiatočný Hyperuzol. Namiesto toho sú rekurencie zoradené do zoznamu od najvyššieho $RecMII$. Ak majú dve rekurencie spoločný uzol, sú spojené do jednej a hodnota $RecMII$ je nastavená na hodnotu tej s vyšším $RecMII$. Pre začiatočný Hyperuzol sa vyberie podgraf, ktorý reprezentuje rekurenciu s najvyšším $RecMII$. Namiesto predošlého algoritmu pre DDG bez rekurencií pokračuje algoritmus nasledovne:

1. Nájsť všetky uzly v grafe na ceste od súčasnej rekurencie (Hyperuzlu) k ďalšej rekurencii v zozname.
2. Redukovať rekurenciu, uzly z predošlého kroku a súčasný Hyperuzol do nového Hyperuzlu.
3. Uzly v Hyperuzle sú topologicky zoradené a pridané do konečnej fronty.
4. Tieto kroky sú opakované dokým nie je graf redukovaný až na acyklický graf bez rekurencií.
5. Už spomenutý algoritmus pre grafy bez rekurencií.

Finálny zoradený zoznam inštrukcií je potom plánovaný. Je tu snaha plánovať inštrukcie čo najbližšie k ich predchodcom a následníkom. Týmto spôsobom redukuje počet potrebných registrov, preto je táto metóda používaná aj v iných metódach, menovite *SMS* (*Swing Modulo Scheduling*). Ak algoritmus nenájde voľné miesto kam naplánovať inštrukciu, hodnota *II* je inkrementovaná a plánovanie začína odznova.

HRMS nachádza takmer optimálny plán za primeraný čas prekladu. Je podobný algoritmu *SMS*, ale narozdiel od neho neberie do úvahy kritickosť uzlov, preto potrebuje väčšie množstvo registrov ako *SMS*.

3.3.5 Integrated Register Sensitive Iterative Software Pipelining

Integrated Register Sensitive Iterative Software Pipelining (*IRIS*) [7] je ďalšou metódou pre *Modulo Scheduling*. Využíva niektoré modifikované heuristiky metódy *Stage Scheduling* a metódy IMS.

Tento algoritmus sa snaží dosiahnuť čo najnižšej hodnoty *II* a tiež minimálneho využitia registrov. *IRIS* rovnako ako *Slack* plánuje z dvoch smerov, takže niektoré inštrukcie čo najskôr a iné čo najneskôr. Samotný algoritmus je veľmi podobný IMS, ale líši sa niektorými modifikáciami:

- Premenné *estart* a *lstart* sú vypočítané ako popísané v *Slack* metóde.

- Inštrukcie sú plánované čo najskôr alebo čo najneskôr v pláne. Toto rozhodnutie sa vypočíta modifikovaným algoritmom *Stage Scheduling*. Vyhľadáva voľné miesto, kam plánovať inštrukciu od *estart* po *lstart* alebo naopak.

Rovnako ako IMS používa prioritnú frontu na základe výšky v grafe, rovnaký prístup k zvyšovaniu hodnoty *II* aj k spätnému trasovaniu (backtracking). Algoritmus sa líši svojím použitím heuristík *Stage Scheduling*. [9] Tieto heuristiky sa používajú pre zníženie počtu potrebných registrov. V prípade *IRIS* sú tieto heuristiky nasledovné:

- Ak má uzol jedného alebo viac nasledníkov, algoritmus hľadá voľné miesto v pláne od *lstart* po *estart*.
- Ak má uzol iba predchodcov, algoritmus začína od *estart* a hľadá až po *lstart*.
- Ak má inštrukcia iba nasledníkov v čiastočnom pláne, tak je nutné rozhodnúť, či odstránením tejto išntrukcie by sa rozdelil graf na viacero nesúvislých subgrafov. Ak áno, plánuje sa od konca intervalu. To isté sa analyzuje pre predchodcov.
- Ak nepatrí inštrukcia ani do jednej z vyššie popísaných kategórií, začína sa plánovať od *estart*.

Podľa štúdie spomenutej vyššie, *IRIS* našla podobne optimálne plány ako IMS na komplexných architektúrach. *IRIS* metóda bola však hodnotená ako menej efektívna čo sa týka minimalizácie počtu registrov, keď porovnaná so *SMS* metódou a to na všetkých typoch architektúr.

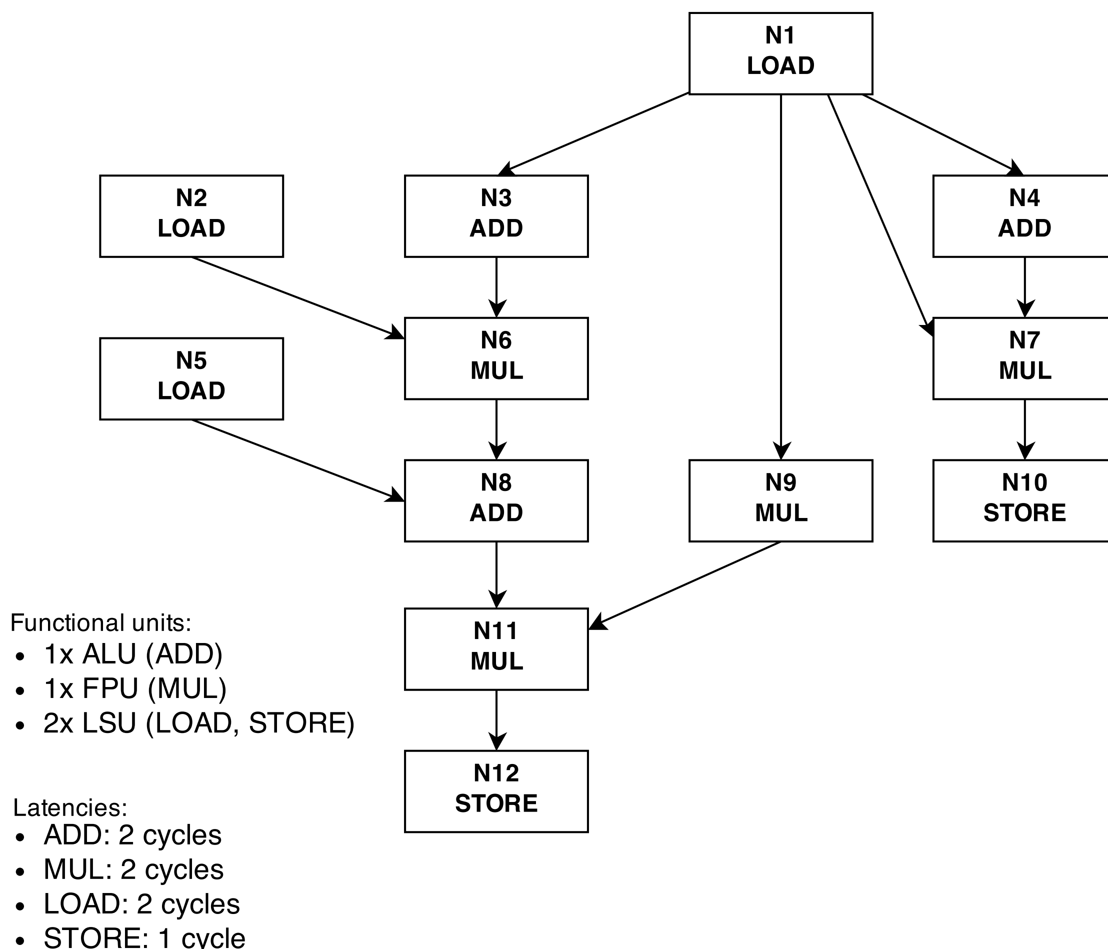
3.3.6 Swing Modulo Scheduling

Swing Modulo Scheduling (SMS) [20] je poslednou z metód otestovaných v spomenutej komparatívnej štúdii. [6] Zistilo sa, že SMS generuje najviac optimálne plány pre jednoduché a stredne zložité architektúry. Tiež dosahuje SMS algoritmus najmenšie využitie registrov pre všetky architektúry a zaberá najmenej času pri preklade. Pre komplexné architektúry, iteratívne metódy (IMS a IRIS) obe našli lepší finálny plán, ale využili pri tom omnoho viac registrov, aj po použití *Stage Scheduling*. Pretože SMS úspešne nachádza takmer optimálny plán za krátky čas a s najmenším počtom registrov, bol vybraný ako cieľ pre túto prácu.

3.4 Swing Modulo Scheduling

Swing Modulo Scheduling (SMS) je ďalším efektívnym heuristickým algoritmom z rodiny *Modulo Scheduling*. Tento algoritmus opäť začína postavením DDG a výpočtom MII. Ďalej počíta niekoľko atribútov pre uzly v grafe, ktoré neskôr určujú poradie plánovania. Algoritmus uprednostňuje rekurencie, keďže tieto majú najväčší vplyv na finálny plán. Preto pri výbere poradia inštrukcií na plánovanie najprv zoraďuje uzly v silno prepojených komponentách (SCC) a podobne ako HRMS najprv plánuje tieto inštrukcie a až potom zvyšok grafu.

Algoritmus je pre jednoduchosť najprv vysvetlený na príklade DDG bez rekurencií a potom je dostupný ďalší príklad, na ktorom sú znázornené zmeny oproti príkladu bez rekurencií. Na obrázku 3.2 je znázornený graf závislostí pre jednu iteráciu, bez rekurencií. V príklade je uvedená hardvérová konfigurácia obsahujúca 1 ADD jednotku, 1 MUL jednotku a dve LSU (*Load-store Units*). Pre jednoduchosť, inštrukcie ADD, MUL, LOAD a STORE môžu byť vykonané len na príslušných jednotkách. Všetky inštrukcie okrem STORE majú latenciu 2 cykly, STORE má latenciu 1 cyklus.



Obrázok 3.2: Príklad grafu dátových závislostí pre SMS

3.4.1 Výpočet MII a atribútov

Prvým krokom tohoto algoritmu je vypočítanie *MII* spôsobom, aký je popísaný v sekcii .

Keďže neexistujú rekurencie v grafe, platí v tomto prípade $RecMII = 0$.

$ResMII$ je vypočítaný ako počtu inštrukcií a jednotiek na ktorých môžu byť vykonávané.

$$ResMII = \max_{\forall r} \left\lceil \frac{N_r}{F_r} \right\rceil = \max\left(\left\lceil \frac{3}{1} \right\rceil, \left\lceil \frac{2+3}{2} \right\rceil, \left\lceil \frac{4}{1} \right\rceil \right) = 4$$

Z toho jednoducho vypočítame MII:

$$MII = \max(RecMII, ResMII) = \max(0, 4) = 4$$

SMS algoritmus potrebuje na svoje správne fungovanie vypočítať okrem MII ďalšie atribúty pre každý jednotlivý uzol. Na základe týchto atribútov sú potom zoradené do prioritnej fronty pre konečné plánovanie. Pre výpočet týchto atribútov sa zvyčajne dočasne z grafu odstránia rekurencie tým, že jedna z hrán tejto rekurencie sa ignoruje. V ďalších

výpočtoch sú označenia λ_u pre latenciu jedného uzlu a $\delta_{u,v}$ pre iteračnú vzdialenosť dvoch uzlov v grafe. $Pred(u)$ označuje množinu predchodcov uzlu a $Suc(u)$ označuje množinu následníkov. Celkovo algoritmus potrebuje vypočítať nasledujúce atribúty:

- **ASAP** (*As Soon As Possible*) určuje najskoršie miesto v pláne, kedy môže byť inštrukcia naplánovaná. Ak nemá predchodcov, je nulová a ak áno, výpočet je nasledovný:

$$ASAP_u = \max(ASAP_v + \lambda_v - \delta_{v,u} \times MII) \forall v \in Pred(u) \tag{3.1}$$

- **ALAP** (*As Late As Possible*) určuje naopak miesto, kde môže byť inštrukcia najneskôr. Ak má uzol následovníkov, je použitá rovnica 3.2. Inak je použitá rovnica 3.3.

$$ALAP_u = \max(ASAP_v) \forall v \in V \tag{3.2}$$

$$ALAP_u = \min(ALAP_v - \lambda_u + \delta_{u,v} \times MII) \forall v \in Suc(u) \tag{3.3}$$

- **MOV** (*Mobility*) alebo pohyblivosť určuje voľnosť inštrukcie pri plánovaní, teda počet cyklov, na ktoré môže byť inštrukcia naplánovaná. Je to hodnota obdobná hodnote *Slack*. 3.3.3

$$MOV_u = ALAP_u - ASAP_u \tag{3.4}$$

- **D** (*Depth*) určuje hĺbku uzlu v DDG. Je definovaný hĺbkou svojich predchodcov a ich latenciami. Ak uzol nemá predchodcov, táto hodnota je nulová. Inak sa počíta ako:

$$D_u = \max(D_v + \lambda_v) \forall v \in Pred(u) \tag{3.5}$$

- **H** (*Height*) určuje výšku uzlu. Analogicky ako u hĺbky, ak uzol nemá následníkov, je výška nulová. Inak je výpočet nasledovný:

$$H_u = \max(H_v + \lambda_u) \forall v \in Suc(u) \tag{3.6}$$

V tabuľke 3.1 sú vypočítané hodnoty atribútov pre jednotlivé uzly.

3.4.2 Zoraďovanie uzlov

Fáza zoraďovania uzlov berie ako vstup DDG a vyššie spočítané atribúty. Z nich vypočíta zoznam inštrukcií zoradených tak, aby z nich bolo možné vytvoriť finálny plán. Fáza plánovania potom najprv alokuje miesto v pláne pre prvú inštrukciu v zozname, potom hľadá miesto pre ďalšiu s tým, že rešpektuje miesto zabrané prvou inštrukciou. Ak nie je možné tento plán vytvoriť, teda neexistuje miesto, kam môže byť inštrukcia naplánovaná, je navýšená hodnota *II*.

Cieľom zoraďovania inštrukcií je hlavne dať prioritu inštrukciám lokalizovaným na kritických cestách v grafe. Inštrukcie, ktoré sa plánujú neskôr, majú menší priestor, kam sa dajú naplánovať, takže je to kompenzované vyššiou pohyblivosťou (MOV_u). Preto jednou z priorít, podľa ktorých sa zoraďujú inštrukcie je aj táto hodnota. Využitie tohoto prístupu má tendenciu znižovať hodnotu *II* a tým nachádzať lepší plán.

Uzol	ASAP	ALAP	MOV	D	H	Jednotka
N1	0	0	0	0	10	LSU
N2	0	2	2	0	8	LSU
N3	2	2	0	2	8	ADD
N4	2	6	4	2	4	ADD
N5	0	4	4	0	6	LSU
N6	4	4	0	4	6	MUL
N7	4	8	4	4	2	MUL
N8	6	6	0	6	4	ADD
N9	2	6	4	2	4	MUL
N10	6	10	4	6	0	LSU
N11	8	8	0	8	2	MUL
N12	10	10	0	10	0	LSU

Tabuľka 3.1: Atribúty jednotlivých uzlov z grafu 3.2

Ďalším cieľom je redukovať hodnotu *MaxLive*, ktorá reprezentuje maximálny počet potrebných hodnôt v jeden časový okamih v pláne. Preto sa využíva podobná technika ako v metóde *HRMS*, čiže plánovanie čo najbližšie k svojím predchodcom a následníkom.

V prípade, že graf neobsahuje rekurencie, intuitívne sa prechádza graf závislostí najprv zdola od uzlu s najväčšiou hĺbkou *D*. V prípade rovnakej hĺbky sa sekundárne vyberá podľa vyššej pohyblivosti. V uvedenom príklade má najvyššiu hĺbku práve uzol *N12*, takže sa začína s poradím:

`O = N12`

Potom algoritmus prechádza všetkých predchodcov tohoto uzlu a vyberá ďalšie uzly primárne podľa hĺbky a sekundárne podľa pohyblivosti. Keď sa algoritmus dostane do stavu, keď už nemá žiaden z pridaných uzlov predchodcov, pokračuje sa smerom dolu.[2] Vo vyššie uvedenom príklade vychádza toto poradie ako:

`O = {N12, N11, N8, N6, N3, N9, N1, N2, N5}`

Ďalej algoritmus pokračuje v grafe smerom dolu a tentokrát je prioritou nižšia výška *H*. Pri rovnakej výške je opäť využitá hodnota MOV_u. Týmto vzniká nasledovné poradie uzlov:

`O = {N12, N11, N8, N6, N3, N9, N1, N2, N5, N4, N7, N10}`

Toto striedanie smerov a následné zoraďovanie uzlov v týchto smeroch (tzv. *swinging*) je vykonávané až dokým nie sú zoradené všetky uzly v grafe.

Pre graf s rekurenciami je zoraďovanie o niečo zložitejšie. Pozostáva z dvoch fáz, kedy algoritmus najprv vytvorí prioritné fronty uzlov pre každú rekurenciu a tie ešte zoraďuje. Je to bližšie vysvetlené na ďalšom príklade grafu s rekurenciami. 3.3.

[2] Z tohoto striedania smerov sa práve algoritmus volá **Swing** Modulo Scheduling.

3.4.3 Plánovanie

Keď je k dispozícii finálny zoznam inštrukcií v poradí, v akom budú plánované, ďalšiou fázou je samotné plánovanie. Plánovanie pozostáva vlastne z mapovania inštrukcií na jednotlivé cykly procesoru a funkcionálne jednotky. Vytvára teda tabuľku aká je zobrazená na **??**. Jeden riadok je potom *bundle*, balík inštrukcií vykonaných paralelne.

Plánovanie analyzuje uzly v zozname podľa ich poradia a snaží sa vytvoriť plán tak, aby boli operácie čo najbližšie svojím predchodcom a následníkom. Preto musí najskôr analyzovať uzol a zoznam, a potom plánovať rôznymi spôsobmi.

- Ak má uzol u len predchodcov vo finálnom pláne, je operácia naplánovaná čo najskôr. Ak sú už obsadené sloty v *MRT* inými inštrukciami, postupuje od cyklu $ASAP_u$ až po cyklus $ALAP_u$ kým nenájde voľný slot. Inštrukcia je vložená do finálneho plánu a slot v *MRT* sa označí ako obsadený.

- Analogicky ak má uzol u len následníkov, je operácia plánovaná čo najneskôr. Opäť kontroluje *MRT*, aby sa vyhol konfliktom a hľadá voľný slot začínajúc od $ALAP_u$ a končiac pri $ASAP_u$.

- V prípade, že má inštrukcia vo finálnom pláne aj predchodcov aj následníkov (toto sa vzhľadom na povahu algoritmu stane iba pre uzol v rekurencii), algoritmus ho radí ako keby mal len predchodcov a hľadá plán od $ASAP_u$.

- Ak nemá uzol u predchodcov ani následníkov, je plánovaný od $ASAP_u$ po $ALAP_u$.

Ak nie je nájdený ani jeden voľný slot pre inštrukciu, znamená to, že algoritmus nevie nájsť finálny plán. Plánovanie je ukončené a inkrementuje sa hodnota *II*. Plán je zahodený a plánovanie sa začína odznova s novým *II*.

Pre uvedený príklad je týmto spôsobom nájdený finálny plán. Ako prvý je nájdený slot pre uzol N12. Keďže nemá ani predchodcov ani následníkov, je naplánovaný na $ASAP_{N12}$. Pokračuje sa po vyššie uvedenom zozname `O = {N12, N11, N8, N6, N3, N9, N1, N2, N5, N4, N7, N10}` až kým nie je naplánovaná každá inštrukcia. Tento plán je uvedený v 3.2. Obsadenosť *MRT* jednotlivými uzlami je zobrazená na 3.3.

Pre názornosť je tabuľka 3.2 rozdelená po *II* cykloch na tri úseky. Tieto úseky sa vykonávajú postupne, takže v cykle 4, keď začína vykonávanie druhého úseku v pláne sa začne tiež vykonávať prvý úsek novej iterácie. Táto začiatočná časť, ktorá ešte nevykonáva všetky iterácie a nie je naplno využitý *Software Pipelining*, je *prológ*.

Vykonávaním tretej iterácie sa teda vykonávajú všetky tri iterácie naraz ako znázornené v *MRT*. Toto je kernel, ktorý sa vykonáva po väčšinu iterácií.

Po ukončení tretej iterácie od konca sa musia stále dve iterácie dokončiť. Vtedy nastupuje ukončenie *kernelu* a vykonáva sa *epilóg*. *Epilóg* pozostáva z dokončenia vykonávania posledných dvoch iterácií, teda druhého a tretieho úseku, ktoré sa vykonajú paralelne a ukončenie poslednej iterácie posledným úsekom. Vykonávanie týchto častí bolo už znázornené vyššie na obrázku 3.1.

3.4.4 Zoraďovanie uzlov v Grafe obsahujúcom rekurencie

V prípade, že graf obsahuje rekurencie, je algoritmus o niečo zložitejší. Beriem do úvahy graf 3.3 s štyrmi univerzálnymi funkcionálnymi jednotkami. Inštrukcie majú všetky latenciu dva cykly. Graf obsahuje dve rekurencie, R_1 z uzlov $\{A, C, D, F\}$ a iteračnou vzdialenosťou medzi F, A rovnú jednej iterácii. Druhou rekurenciou je R_2 skladajúca sa

Cyklus	LSU 1	LSU 2	ADD	MUL
0	N1			
1				
2	N2		N3	
3				
4			N4	N6
5		N5		
6				N7
7			N8	N9
8		N10		
9				N11
10		N12		
11				

Tabuľka 3.2: Plán inštrukcií pre príklad 3.2

Cyklus	LSU 1	LSU 2	ADD	MUL
0	N1	N10	N4	N6
1		N5		N11
2	N2	N12	N3	N7
3			N8	N9

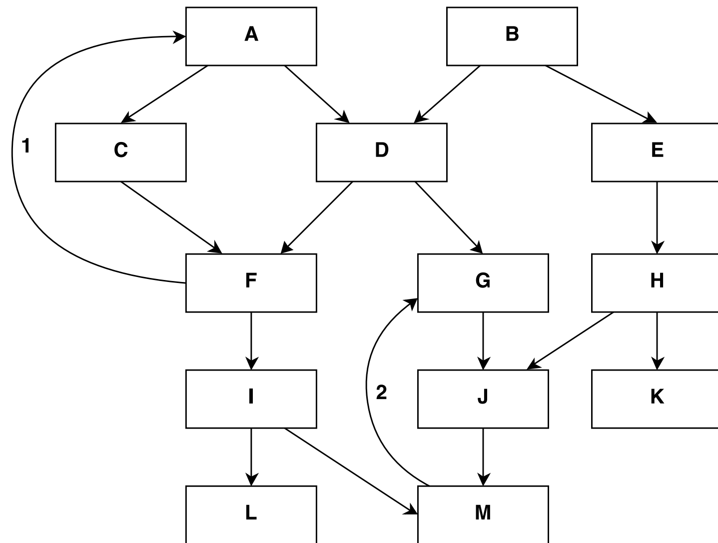
Tabuľka 3.3: *MRT* (*Modulo Reservation Table*) pre príklad 3.2

z uzlov $\{G, J, M\}$ s iteračnou vzdialenosťou medzi H, J dve iterácie. Keďže v tomto príklade vychádza $RecMII_{R1} = 6$ a $RecMII_{R2} = 3$, začína algoritmus s prvou rekurenciou.

Zoraďovať sa nezačína od najvyššej hĺbky v grafe, ale od rekurencií s najvyšším *RecMII*. Je to algoritmus veľmi podobný tomu ktorý je použitý v *HRMS*.

Graf je najprv rozdelený do niekoľkých množín podľa počtu rekurencií. Tými sú:
$S_1 = \{A, C, D, F\}$
$S_2 = \{G, J, M, I\}$
$S_2 = \{B, E, H, K, L\}$
V množine S_1 je len prvá rekurencia s najvyšším *RecMII*. V množine S_2 je započítaná druhá rekurencia a zároveň všetky uzly, ktoré sú na ceste medzi prvou a druhou rekurenciou. Zabraňuje sa tým stavu, kedy by boli naplánovaní aj predchodci aj nasledníci uzlu predtým, ako by bol uzol sám naplánovaný. Ak by existovala ďalšia rekurencia, tvorila by ďalšiu množinu uzlov, ktoré obsahujú túto rekurenciu a všetky uzly medzi rekurenciami. Nakoniec množina S_3 obsahuje zvyšok uzlov v grafe, ktoré sú mimo rekurencií.

Druhou časťou tohoto zoraďovacieho algoritmu je zoradiť uzly v týchto množinách. Pri tomto zoraďovaní sa ignoruje mimo-iteračná závislosť a tento podgraf je zoradený podľa hĺbky D_u a sekundárne podľa mobility MOV_u. Vzniká postupne finálny plán zoraďovaním jednotlivých množín. Najprv sa vytvorí čiastočný plán z prvej množiny:
$O = < F, C, D, A >$



Obrázok 3.3: Príklad DDG s rekurenciami, inšpirovaný [20].

Keďže pôvodný smer grafom je zdola nahor, algoritmus hľadá predchodcov tejto rekurencie a zisťuje, že žiaden uzol z množiny S_2 nie je predchodcom. Mení smer (*swinging*) a nachádza postupne následníkov rekurencie a tým všetky uzly z množiny S_2.
$O =< F, C, D, A, G, I, J, M >$
Algoritmus pokračuje ďalej predchodcami S_1 a S_2 a nachádza postupne uzly H, E a B.:
$O =< F, C, D, A, G, I, J, M, H, E, B >$
Nakoniec znovu mení smer vyhľadávania a nachádza posledné dva uzly L a K ako následníkov, zoradených podľa výšky. Tým vzniká konečné zoradenie uzlov:
$O =< F, C, D, A, G, I, J, M, H, E, B, L, K >$

Následné plánovanie prebieha rovnako ako pri grafe bez rekurencií v prvom príklade.

Kapitola 4

Návrh a implementácia v LLVM

Ako bolo spomenuté v predchádzajúcich kapitolách, rozhodol som sa implementovať algoritmus *Swing Modulo Scheduling* (ďalej len SMS). Návrh samotného algoritmu a jeho fungovania je popísaný vyššie v sekcii 3.4.

Tento algoritmus už bol implementovaný v skoršej verzii LLVM, dokončený v r.2006, autorkou ktorej je Tanya M. Lattner.[19]. LLVM platforma však odvtedy prešla mnohými zmenami a vo verzii LLVM 1.7 bola odstránená architektúra *SPARC V9*, pre ktorú bola implementácia primárne napísaná. S ňou sa odstránil aj algoritmus pre SMS. LLVM má v súčasnosti (verzie 3.x) značne odlišnú štruktúru, keďže prešla za posledné desaťročie mnohými úpravami. Taktiež existuje implementácia rovnakého algoritmu v GCC od verzie 4.0. [12]

Vzhľadom na existujúcu implementáciu sa zdalo redundantné implementovať celý algoritmus znovu. Inšpiroval som sa preto existujúcou implementáciou v staršej verzii LLVM.

4.1 Závislosť na architektúre

Okrem informácií o prekladanom kóde pre vytvorenie DDG potrebuje plánovač pre správne fungovanie informácie o cieľovej architektúre. Tými sú:

- Počet jednotlivých funkcionálnych jednotiek VLIW, na ktorých je možné vykonávať inštrukcie. Je to obmedzenie, z ktorého sa tiež počíta *ResMII* a bez neho nie je možné korektne vytvárať *bundle* inštrukcií a tým ani vytvoriť správny plán.
- Triedy jednotlivých inštrukcií – konkrétne na ktoré funkcionálne jednotky sa dajú jednotlivé inštrukcie mapovať. Bez tejto informácie nie je možné vedieť, či je možné v jednom cykle mapovať tieto inštrukcie.
- Počet dostupných registrov a informácia o tom, ktoré registre sú univerzálne a ktoré slúžia na špeciálne účely. Je to nutné, keďže algoritmus potencionálne zvyšuje počet potrebných registrov a musí vedieť, či je možné pracovať s toľkými registrami alebo vytvoriť horší plán s menším počtom registrov.

Z tohoto dôvodu sa spravidla zaraďuje *Modulo Scheduling* medzi algoritmy závislé na architektúre. Vďaka návrhu LLVM je však možné tieto informácie získať v rámci prekladu skôr a je možné implementovať algoritmus pre celú triedu architektúr. Keďže sa v Codasip Frameworku generujú modely architektúr, musia byť tieto informácie dostupné aj v týchto modeloch.

4.2 Umiestnenie v LLVM

Bolo nutné rozhodnúť, kde bude implementácia umiestnená a v ktorej fáze prekladu bude algoritmus prebiehať. Tiež bolo potrebné zistiť, či je možné implementáciu previesť do tak odlišnej štruktúry LLVM. V retrospektíve som zistil, že prevod tejto implementácie do novej verzie LLVM zabralo viac času a energie než by pravdepodobne bolo vydané na implementovanie celého algoritmu od základov.

Vzhľadom na fungovanie algoritmu nemohol byť navrhnutý v optimalizátore nad LLVM IR, keďže LLVM v tom stave nepozná inštrukcie cieľovej architektúry a ani ich latencie.

Algoritmus je závislý od architektúry a potrebuje pre správne fungovanie informácie o cieľových inštrukciách a funkcionálnych jednotkách architektúry. Toto implikovalo implementáciu v backende LLVM, ktorý sa skladá z analyzačných a transformačných priechodov, ktoré postupne prebiehajú nad prekladaným kódom. Konkrétne musela byť implementácia až po namapovaní LLVM IR na cieľové inštrukcie, čiže v generátore kódu.

Pre správne fungovanie je nutné mať finálny plán inštrukcií pre cieľovú architektúru, inak môžu ďalšie transformačné priechody nad kódom cieľový plán zmeniť a môže produkovať ešte horší plán než bez prebehnutia SMS. Bolo teda nutné priechod umiestniť čo možno najneskôr.

Na druhú stranu vyvstáva otázka, či priechod môže prebiehať po alokácii registrov (Register Allocation, ďalej len RA). Keďže alokácia má okrem iného za úlohu redukovať počet potrebných registrov, používa často rovnaký register pre viac rôznych premenných. Opätovným použitím registrov vznikajú v grafe *WAR* a *WAW* závislosti, ktoré limitujú možnosti SMS plánovača. V prípade potrebných viacerých registrov pre cieľový plán inštrukcií by tiež bolo nutné dodatočne alokovať registre. Ak by to nebolo možné takto implementovať v LLVM, algoritmus by ešte viac utrpel.

Ak je SMS plánovač umiestnený pred RA, plánovač pracuje s virtuálnymi registrami a stále má k dispozícii informáciu o počte dostupných registrov. Tým môže byť plánovač rozšírený o vyššie spomenutý *Modulo Variable Expansion* a vyrieši sa problémy s prelínaním životnosti premenných v registroch. Toto by pravdepodobne nebolo možné, príp. by to bolo veľmi náročné po *RA*. Priechod je teda umiestnený po zlučovaní registrov (*Register Coalescing*) kvôli získaniu finálneho plánu a tesne pred *RA*.

4.3 Podpora LLVM

Vzhľadom na umiestnenie priechodu musí táto optimalizácia splňovať niektoré požiadavky v LLVM, aby bolo zaistené korektné prevedenie ďalších optimalizácií. Medzi inými je tento priechod až po výpočte životnosti premenných a registrov (*Live Intervals Analysis, Live Variable Analysis, Slot Indexes*). Životnosť premennej začína jej definíciou a končí jej posledným použitím. Medzi týmito intervalmi je premenná „živá“. Keďže alokácia registrov tieto informácie potrebuje pre korektné spracovanie kódu, akékoľvek priechody pridané na toto miesto musia tieto informácie zachovať alebo znovu vypočítať.

Ako som zistil pri implementácii, súčasný stav LLVM poskytuje len veľmi málo nástrojov na aktualizáciu týchto informácií. Táto fáza prekladača je v aktívnom vývoji a očakáva sa, že aktualizácia *LiveIntervals* aj *MI Scheduler*, ktorý pracuje na tejto úrovni sa bude ešte významne meniť a rozširovať. Existuje len málo dokumentácie o súčasnom stave [1] a dostupné utility pre aktualizáciu tiež nie sú podľa developerov plne funkčné

[1]Časť oficiálnej dokumentácie LLVM 3.4 zaoberajúca sa optimalizáciami nad SSA v generátori kódu

a mnohé chýbajú. Príkladom je aktualizácia intervalov pre novo-vytvorené základné bloky potrebné pre vytvorenie prológu a epilógu. Moja implementácia sa venovala verzii LLVM 3.2, s ktorou Codasip Framework v čase začatia tohoto projektu pracoval. Neskôr počas písania tejto práce (Apríl 2014) sa v Codasip Frameworku prešlo na novú verziu LLVM 3.4. Táto verzia má väčšiu podporu pre aktualizáciu *LiveIntervals* a obsahuje niektoré nové triedy plánovačov, ktoré sa aktualizácii *LiveIntervals* venujú, menovite napríklad trieda *ScheduleDAGMILive*.

Ďalší problém je s tvorbou samotných *bundle*. Momentálne má LLVM podporu čiastočnú podporu pre *bundle* tým, že prekladač vníma skupinu inštrukcií ako jednu inštrukciu s väčším množstvom operandov. Podpora *bundle* je však v súčasnosti iba pre priechody až po alokácii registrov a pre priechody pred RA nie je spoľahlivo otestovaná. Z tohoto dôvodu je nutné len označiť inštrukcie, ktoré majú byť vložené do *bundle* a neskôr v ďalšom priechode ich tam vložiť. V LLVM 3.4 bola tiež pridaná podpora aktualizácie intervalov pri presune inštrukcie do *bundle*.

4.4 Implementácia

Algoritmus je implementovaný ako transformačný priechod v generátori kódu v backende LLVM, tesne pred alokáciou registrov. Pracuje postupne nad všetkými základnými blokmi, ktoré najprv kontroluje, či sú telom cyklu a či neobsahujú riadiace štruktúry. Ak nie, pokračuje k vykonianiu SMS priechodu pre tento blok.

Je tu dostupný *Machine Instruction Scheduler*, ktorý plánuje inštrukcie v základných blokoch formou algoritmu *List Scheduling*. Pre detekovanie závislostí využíva klasický DAG, bez podpory cyklických závislostí nad iteráciami.

Vzhľadom na potrebu závislostí medzi iteráciami by bolo nutné upraviť tento graf na cyklický, čo by v dnešnom stave LLVM bolo značne náročné. Upravil som preto graf v existujúcej implementácii SMS, ten je dostupný v *MSchedGraph.h*. Jednotlivé vrcholy grafu sú inštrukcie s pridanými informáciami vrátane latencie a typu inštrukcie. Sú tiež dostupné doplňujúce informácie o hodnotách ASAP, ALAP, MOV, výške a hĺbke v grafe, spočítané v prvej fáze algoritmu.

Algoritmus postupuje tak, ako je popísané v sekcii 3.4, teda vytvorí graf závislostí a následne vypočíta MII z informácií dostupných od architektúry. Ďalej prechádza grafom a vypočíta atribúty potrebné pre ďalšie zoraďovanie. Celý tento postup je popísaný na príklade v návrhu algoritmu.

Algoritmus nájde silno prepojené komponenty a tieto zoraďuje do zoznamu inštrukcií, tým vytvorí čiastočný plán, inštrukcie zoradené podľa toho, ako sa budú plánovať. Inštrukcie plánuje až dokým nenájde finálny plán. V prípade, že tento plán nenájde, zvyšuje hodnotu *II* a opäť opakuje algoritmus. Pri príliš vysokej hodnote *II* sa algoritmus ukončí a tým neurobí žiadne zmeny v cykle.

Algoritmus končí tým, že tvorí dva nové základné bloky a transformuje súčasný blok. Dva vytvorené bloky sú transformované do prológu a epilógu a sú pridané nepodmienené skoky na ich konce. Inštrukcie sú postupne kopírované z originálneho bloku, ktorý obsahuje telo cyklu. Originálne telo cyklu je potom zoradené do podoby kernelu. Vzhľadom na neexistujúcu podporu *bundles* pred *RA* nie sú inštrukcie zatiaľ plánované paralelne.

---

pozostáva zo slov „To Be Written“.

4.5 Výsledky

Vzhľadom k nedostupnosti optimálneho plánovača sa nedá odhadnúť, ako blízko k optimálnym sú nachádzané plány. Môžem vychádzať iba z štúdií vykonaných v minulosti, ktoré tvrdia, že tento algoritmus produkuje takmer optimálny plán vo väčšine prípadov. Ako očakávané, algoritmus zvýšil počet potrebných registrov. Celkovo pre testované programy algoritmus nachádza dostatočne optimálne plány. Je tu viditeľné zlepšenie oproti nezreťazeným cyklom, a to bez dátových konfliktov a za zlomok kompilačného času. Je tu nutné doplniť *Modulo Variable Expansion* kvôli vzniknutým hazardom, inak je algoritmus plne funkčný a nachádza finálny plán pre zreťazenie cyklu. Algoritmus tiež nie je optimalizovaný a momentálne zaberá priemerne 30-40% systémového času pri preklade.

Výsledné fungovanie algoritmu je uvedené na nasledujúcom príklade, kde je ako vstupný súbor jednoduchý cyklus v jazyku C:

```
int a[100], b[100];

int main()
{
    for (int i = 1; i < 100; i++) {
        a[i] = b[i] + 3;
    }
}
```

Tento kód sa transformuje do LLVM IR kódu a následne do výsledných inštrukcií architektúry. Priechod Modulo Scheduling teda pracuje s nasledovnými blokmi[2]:

```
BB#0: derived from LLVM BB %entry
%vreg13<def> = i_select___reg__cond_slez__reg0__op_imm__imm5__op_imm__imm5__ -1;
%vreg3<def> = i_lui___reg__uimm16__ <ga:@a>[TF=3];
%vreg4<def> = i_ori___reg__uimm16__ %vreg3, <ga:@a>[TF=2];
%vreg6<def> = i_lui___reg__uimm16__ <ga:@b>[TF=3];
%vreg7<def> = i_ori___reg__uimm16__ %vreg6, <ga:@b>[TF=2];
%vreg11<def> = i_addi__reg0__imm16__ -100;

BB#1: derived from LLVM BB %for.body
%vreg5<def> = i_sub___reg__src2_am_shl2___reg__reg0__ %vreg4, %vreg13;
%vreg8<def> = i_sub___reg__src2_am_shl2___reg__reg0__ %vreg7, %vreg13;
%vreg9<def> = i_ld___reg__imm16__SPEC_CLONE_ %vreg8;
%vreg10<def> = i_addi___reg__imm16__ %vreg9, 3;
i_st___reg___reg__imm16__SPEC_CLONE_ %vreg10, %vreg5;
%vreg13<def> = i_addi___reg__imm16__ %vreg13, -1;
i_jump_cond__cond_ne___reg__op_reg___reg__rel_addr14__ %vreg13, %vreg11, <BB#1>;
i_jump_call_abs__opc_jumpi__addr26__ <BB#2>

BB#2: derived from LLVM BB %for.end
%vreg12<def> = i_subi__reg0__imm16__SPEC_CLONE_;
%regs_3<def> = COPY %vreg12;
```

[2]Názvy inštrukcií sú o niečo skrátené pre lepšiu čitateľnosť.

```
i_jump_reg__opc_jump___reg__RET_CLONE_ %regs_31<kill>, %regs_3<kill>
```

Vo funkcii sú pre jednoduchosť príkladu iba tri základné bloky: začiatok, telo a ukončenie cyklu. Keďže Modulo Scheduling transformuje iba telo cyklu, zameriam sa len naň. Toto telo je priechodom transformované na tri základné bloky ako bolo popísané vyššie, na prológ, kernel a epilóg. Kernel je novým telom cyklu, ktorý prelína iterácie.

Zdanlivo sa jedná o veľkú expanziu kódu, pretože inštrukcie nie sú v *bundle*. Taktiež nie je predvedené striedanie registrov vo forme *Modulo Variable Expansion*, pre toto je nutné rozbaliť kernel a alokovať niekoľkonásobné množstvo registrov. Len v tomto jednoduchom príklade je nutných päťkrát viac registrov, keďže v kerneli je vykonávaných naraz päť iterácií. Ak neexistuje dostatočný počet registrov, je nutné zvýšiť *II* a vytvoriť menej optimálny plán, ktorý na druhú stranu využíva menej registrov.

```
BB#3: derived from LLVM BB %Prologue
%vreg5<def> = i_sub___reg__src2_am_shl2___reg__reg0__ %vreg4, %vreg13;
%vreg8<def> = i_sub___reg__src2_am_shl2___reg__reg0__ %vreg7, %vreg13;
%vreg5<def> = i_sub___reg__src2_am_shl2___reg__reg0__ %vreg4, %vreg13;
%vreg8<def> = i_sub___reg__src2_am_shl2___reg__reg0__ %vreg7, %vreg13;
  %vreg13<def> = i_addi___reg__imm16__ %vreg13, -1;
%vreg9<def> = i_ld___reg__imm16__SPEC_CLONE_ %vreg8;
%vreg5<def> = i_sub___reg__src2_am_shl2___reg__reg0__ %vreg4, %vreg13;
%vreg8<def> = i_sub___reg__src2_am_shl2___reg__reg0__ %vreg7, %vreg13;
%vreg13<def> = i_addi___reg__imm16__ %vreg13, -1;
%vreg9<def> = i_ld___reg__imm16__SPEC_CLONE_ %vreg8;
%vreg5<def> = i_sub___reg__src2_am_shl2___reg__reg0__ %vreg4, %vreg13;
%vreg8<def> = i_sub___reg__src2_am_shl2___reg__reg0__ %vreg7, %vreg13;
%vreg10<def> = i_addi___reg__imm16__ %vreg9, 3;
%vreg13<def> = i_addi___reg__imm16__ %vreg13, -1;
%vreg9<def> = i_ld___reg__imm16__SPEC_CLONE_ %vreg8;
i_jump_call_abs__opc_jumpi__addr26__ <BB#1>

BB#1: derived from LLVM BB %for.body
%vreg5<def> = i_sub___reg__src2_am_shl2___reg__reg0__ %vreg4, %vreg13;
%vreg8<def> = i_sub___reg__src2_am_shl2___reg__reg0__ %vreg7, %vreg13;
%vreg10<def> = i_addi___reg__imm16__ %vreg9, 3;
i_st___reg___reg__imm16__SPEC_CLONE_ %vreg10, %vreg5;
%vreg13<def> = i_addi___reg__imm16__ %vreg13, -1;
%vreg9<def> = i_ld___reg__imm16__SPEC_CLONE_ %vreg8;
i_jump_cond_ne___reg__op_reg___reg__rel_addr14__ %vreg13, %vreg11, <BB#1>;
i_jump_call_abs__opc_jumpi__addr26__ <BB#4>

BB#4: derived from LLVM BB %Epilogue
%vreg10<def> = i_addi___reg__imm16__ %vreg9, 3;
i_st___reg___reg__imm16__SPEC_CLONE_ %vreg10, %vreg5;
%vreg13<def> = i_addi___reg__imm16__ %vreg13, -1;
%vreg9<def> = i_ld___reg__imm16__SPEC_CLONE_ %vreg8;
%vreg10<def> = i_addi___reg__imm16__ %vreg9, 3;
i_st___reg___reg__imm16__SPEC_CLONE_ %vreg10, %vreg5;
```

```
%vreg10<def> = i_addi___reg__imm16__ %vreg9, 3;
i_st___reg___reg__imm16__SPEC_CLONE_ %vreg10, %vreg5;
i_st___reg___reg__imm16__SPEC_CLONE_ %vreg10, %vreg5;
i_jump_call_abs__opc_jumpi__addr26__ <BB#2>
```

Algoritmus sa dá otestovať z viacerých pohľadov a to vzhľadom na počet využitých registrov, využitie *bundle* a funkcionálnych jednotiek architektúry, rýchlosť kompilácie a rýchlosť výsledného programu v porovnaní s programom preloženým bez vykonania priechodu.

Kapitola 5

Možnosti ďalšej práce

Pre pokračovanie v tejto práci je nutné vyriešiť konflikt s ďalšími plánovačmi pre VLIW, menovite *PreEmitVLIWScheduling*. Je nutné pridať do alokácie registrov variantu *Modulo Variable Expansion* kvôli konfliktom pri reťazení iterácií a tým použití väčšieho množstva registrov.

Navrhujem tiež rozšíriť DAG triedy *ScheduleDAGMILive*, ktorý sa používa v *MI Scheduleri* o cyklické závislosti a hrany v grafe rozšíriť o vlastnosť „vzdialenosti" v počte iterácií, ak to je možné. Použitý graf síce v súčasnosti správne reprezentuje DDG, ale vzhľadom na súčasný vývoj LLVM a nekompatibilitu rozpoznávania hazardov s touto implementáciou by bolo vhodné previesť implementáciu na tento LLVM štandard. LLVM obsahuje množstvo utilít, ktoré nebolo možné použiť kvôli tomuto nekompatibilnému grafu a z neho vychádzajúcich štruktúr. Vyriešením tohoto problému by sa implementácia algoritmu značne zjednodušila.

Boli tiež navrhnuté globálne metódy plánovania pre zreťazenie cyklov. Jedná sa o metódy pre komplexné cykly, ktoré zostávajú nezreťazené bežným prístupom cyklického plánovania. Okrem globálneho prístupu *Modulo Scheduling* je tu viacero metód pre dosiahnutie cieleného zreťazenia ako *Perfect Pipelining* [2], *GURPR** [30] a *Enhanced Pipelining* [15].

Tiež existujú globálne metódy triedy *Modulo Scheduling*, ktoré transformujú komplexný cyklus do jedného základného bloku. Generovanie výsledného kódu je potom zložitejšie, keďže sa musí zaoberať Vzhľadom na existujúci *Superblock Scheduling* je tiež možné rozšíriť *Modulo Scheduling* na superbloky, ako popísané v práci T.M.Lattner [19]. Toto riešenie ďalej zvyšuje paralelizmus na úrovni inštrukcií a zlepšuje výsledný plán.

Pre upravenie Modulo Scheduling pre komplexné cykly je potrebné pridať jednu z metód pre hierarchickú redukciu. Tieto metódy dovoľujú reťaziť cykly tak, že najprv naplánujú najhlbší cyklus a potom berú tento vnorený cyklus do úvahy ako jednu makro-inštrukciu. Tým redukujú cyklus na jednoduchý a dá sa použiť originálna metóda Modulo Scheduling. Tak isto sa dajú vyriešiť podmienené príkazy v cykle, môžu sa brať ako jedna inštrukcia.

Tieto metódy by výrazne zvýšili množstvo cyklov, ktoré je možné zreťaziť a tým zlepšili beh preloženého programu a využitie funkcionálnych jednotiek.

Kapitola 6

Záver

Táto práca sa zaoberá metódami pre zreťazenie cyklov (*Software Pipelining*) pre VLIW a superskalárne architektúry. Cieľom práce bolo implementovať optimalizačný priechod, ktorý transformuje telá cyklov tak, aby bolo možné vykonávať viac iterácií zároveň.

Projekt je súčasťou väčšieho projektu *Codasip Framework*, ktorý obsahuje okrem iných nástrojov aj prekladač jazyka C pre modely architektúr. Práca popisuje bližšie umiestnenie a význam projektu v *Codasip Frameworku*. Pre lepšiu pochopiteľnosť popisuje samotný prekladač a generovanie prekladača. Tiež sa venuje samotným architektúram a rozdielom medzi VLIW a superskalárnymi architektúrami. Vzhľadom na množstvo metód pre dosahovanie zreťazenia cyklov sú tu načrtnuté rôzne metódy na plánovanie inštrukcií a nakoniec detailne popísaná zvolená metóda *Swing Modulo Scheduling* a jej implementácia.

Táto metóda dosahuje vo väčšine prípadov takmer optimálny finálny plán s podstatne menším nárastom potrebných registrov ako ostatné techniky. Tento nárast je však stále dostatočne vysoký, takže architektúra musí mať o dosť väčšie množstvo registrov, aby bolo možné cykly zreťaziť optimálne. Je to významná optimalizácia, ktorá výrazne zvyšuje paralelizmus na úrovni inštrukcií za cenu využitia vyššieho počtu registrov.

Implementoval som tento algoritmus ako priechod v backende prekladača LLVM a prebieha tesne pred alokáciou registrov. Priechod transformuje telo cyklu a nachádza finálny plán. Sú navrhnuté rozšírenia algoritmu a existuje potenciál rozširovať tento algoritmus na reťazenie zložitejších cyklov.

Príloha A

Návod na použitie a obsah CD

A.1 Návod

Táto časť popisuje návod na preklad a použitie zdrojových súborov na priloženom CD. Je nutné mať prístup k preloženým a nainštalovaným nástrojom Codasip Framework, ako popísané na https://www.codasip.com/wiki/doku.php?id=navody-k-prekladu. V `readme.txt` je popísaný návod na pridanie zdrojových súborov do emphcompilerdev/ V prípade napríklad architektúry *Codix VLIW* sa najprv vstupný program v jazyku C preloží napríklad pomocou *clang*:

```
./codix_vliw_clang -emit-llvm -S test.c -O3 -o test.ll
```

Potom pre preklad z medzikódu je možné použiť nástroj *llc*. Nástroj slúži ako backend a jeho výstupom je program v jazyku symbolických adries danej architektúry.

```
./codix_vliw_llc -O3 test.ll -o test.asm -print-after-all 2> out.txt # -debug
```

Program je ďalej možné spracovať ďalšími nástrojmi *Codasip Frameworku.* Pre pozorovanie správania sa priechodu SMS však stačí súbor `out.txt`, kde je možné nájsť kód pred a po vykonaní priechodu. Pre viac detailov pre tento výpis je možné pridať parameter `-debug`. Príklad jednoduchého cyklu a jeho prekladu je dostupný na CD v zložke `example/`.

A.2 Obsah CD

Obsahom priloženého CD sú:

- **BP-xglasn00.pdf** – elektronická verzia textu bakalárskej práce
- **BP-xglasn00-src/** – zdrojové súbory k textu bakalárskej práce
- **example/** – príklad cyklu a výsledného plánu s návodom na použitie
- **src/** – zdrojové súbory k implementačnej časti bakalárskej práce

Literatúra

[1] Abraham, S. G.; Kathail, V.; Deitrich, B. L.: Meld scheduling: A technique for relaxing scheduling constraints. *International journal of parallel programming*, ročník 26, č. 4, 1998: s. 349–381.

[2] Aiken, A.; Nicolau, A.: Perfect pipelining: A new loop parallelization technique. In *ESOP'88*, Springer, 1988, s. 221–235.

[3] Allan, V. H.; Jones, R. B.; Lee, R. M.; aj.: Software Pipelining. *ACM Comput. Surv.*, ročník 27, č. 3, Září 1995: s. 367–432, ISSN 0360-0300, doi:10.1145/212094.212131. URL http://doi.acm.org/10.1145/212094.212131

[4] Banerjia, S.; Havanki, W. A.; Conte, T. M.: Treegion scheduling for highly parallel processors. In *Euro-Par'97 Parallel Processing*, Springer, 1997, s. 1074–1078.

[5] Bharadwaj, J.; Chen, W. Y.; Chuang, W.; aj.: The Intel IA-64 compiler code generator. *Micro, IEEE*, ročník 20, č. 5, 2000: s. 44–53.

[6] Codina, J. M.; Llosa, J.; González, A.: A comparative study of modulo scheduling techniques. In *Proceedings of the 16th international conference on Supercomputing*, ACM, 2002, s. 97–106.

[7] Dani, A. K.; Janaki Ramanan, V.; Govindarajan, R.: Register-sensitive software pipelining. In *Parallel Processing Symposium, 1998. IPPS/SPDP 1998. Proceedings of the First Merged International... and Symposium on Parallel and Distributed Processing 1998*, IEEE, 1998, s. 194–198.

[8] Dehnert, J. C.; Hsu, P. Y.-T.; Bratt, J. P.: Overlapped loop support in the Cydra 5. In *ACM SIGARCH Computer Architecture News*, ročník 17, ACM, 1989, s. 26–38.

[9] Eichenberger, A. E.; Davidson, E. S.: Stage scheduling: A technique to reduce the register requirements of a modulo schedule. In *Proceedings of the 28th annual international symposium on Microarchitecture*, IEEE Computer Society Press, 1995, s. 338–349.

[10] Fisher, J. A.: Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, ročník 30, č. 7, 1981: s. 478–490.

[11] Fisher, J. A.; Landskov, D.; Shriver, B. D.: Microcode compaction: looking backward and looking forward. In *Proceedings of the May 4-7, 1981, national computer conference*, ACM, 1981, s. 95–102.

[12] Hagog, M.; Zaks, A.: Swing modulo scheduling for gcc. In *Proceedings of the 2004 GCC Developers' Summit*, 2004, s. 55–64.

[13] Huff, R. A.: Lifetime-sensitive modulo scheduling. In *ACM SIGPLAN Notices*, ročník 28, ACM, 1993, s. 258–267.

[14] Hwu, W.-M. W.; Mahlke, S. A.; Chen, W. Y.; aj.: The superblock: an effective technique for VLIW and superscalar compilation. *the Journal of Supercomputing*, ročník 7, č. 1-2, 1993: s. 229–248.

[15] Jones, R. B.; Allan, V. H.: Software pipelining: an evaluation of enhanced pipelining. In *Proceedings of the 24th annual international symposium on Microarchitecture*, ACM, 1991, s. 82–92.

[16] Lam, M.: Software pipelining: An effective scheduling technique for VLIW machines. In *ACM Sigplan Notices*, ročník 23, ACM, 1988, s. 318–328.

[17] Lattner, C.: The Design of LLVM. http://www.drdobbs.com/architecture-and-design/the-design-of-llvm/240001128#, accessed: 2014-05-05.

[18] Lattner, C.; Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. San Jose, CA, USA, Mar 2004, s. 75–88.

[19] Lattner, T. M.: *An Implementation of Swing Modulo Scheduling with Extensions for Superblocks*. Diplomová práce, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, June 2005, *See* http://llvm.cs.uiuc.edu.

[20] Llosa, J.; González, A.; Ayguadé, E.; aj.: Swing module scheduling: a lifetime-sensitive approach. In *Parallel Architectures and Compilation Techniques, 1996., Proceedings of the 1996 Conference on*, IEEE, 1996, s. 80–86.

[21] Llosa, J.; Valero, M.; Ayguadé, E.; aj.: Hypernode Reduction Modulo Scheduling. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, MICRO 28, Los Alamitos, CA, USA: IEEE Computer Society Press, 1995, ISBN 0-8186-7349-4, s. 350–360.
URL
http://dl.acm.org.ezproxy.lib.vutbr.cz/citation.cfm?id=225160.225211

[22] Mahlke, S. A.; Lin, D. C.; Chen, W. Y.; aj.: Effective Compiler Support for Predicated Execution Using the Hyperblock. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, MICRO 25, Los Alamitos, CA, USA: IEEE Computer Society Press, 1992, ISBN 0-8186-3175-9, s. 45–54.
URL http://dl.acm.org/citation.cfm?id=144953.144998

[23] Mantripragada, S.; Jain, S.; Dehnert, J.: A new framework for integrated global local scheduling. In *Parallel Architectures and Compilation Techniques, 1998. Proceedings. 1998 International Conference on*, Oct 1998, ISSN 1089-795X, s. 167–174, doi:10.1109/PACT.1998.727189.

[24] Mináč, T.: *Kompilátor jazyka C pro VLIW architektury*. Diplomová práce, FIT VUT v Brně, Brno, 2013.

[25] Nicolau, A.: Percolation scheduling: A parallel compilation technique. Technická zpráva, Cornell University, 1985.

[26] Nicolau, A.; Novack, S.: Trailblazing: A hierarchical approach to percolation scheduling. In *ICPP*, Citeseer, 1993, s. 120–124.

[27] Rau, B. R.: Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proceedings of the 27th annual international symposium on Microarchitecture*, ACM, 1994, s. 63–74.

[28] Ruttenberg, J.; Gao, G. R.; Stoutchinin, A.; aj.: Software pipelining showdown: Optimal vs. heuristic methods in a production compiler. In *ACM SIGPLAN Notices*, ročník 31, ACM, 1996, s. 1–11.

[29] Srikant, Y.; Shankar, P.: *The Compiler Design Handbook: Optimizations and Machine Code Generation*. CRC Press, 2007.

[30] Su, B.; Wang, J.: GURPR*: a new global software pipelining algorithm. In *Proceedings of the 24th annual international symposium on Microarchitecture*, ACM, 1991, s. 212–216.