

Datově orientovaný přístup při vývoji software

Bc. Tomáš Janečka

Diplomová práce
2023



Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
Ústav informatiky a umělé inteligence

Akademický rok: 2022/2023

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	Bc. Tomáš Janečka
Osobní číslo:	A20131
Studijní program:	N0613A140022 Informační technologie
Specializace:	Softwarové inženýrství
Forma studia:	Prezenční
Téma práce:	Datově orientovaný přístup při vývoji software
Téma práce anglicky:	Data-Oriented Software Design

Zásady pro vypracování

1. Definujte pojem datově orientovaný návrh a seznamte se s touto problematikou.
2. Porovnejte tento způsob návrhu s objektově orientovaným návrhem.
3. Popište vliv mikroarchitektury počítače na rychlost běhu programu.
4. Demonstrujte jednotlivé principy na příkladech.
5. Ověřte efektivitu programů pomocí nástrojů pro výkonostní testy a profilování.
6. Sestavte sadu doporučení pro využití datově orientovaného přístupu.

Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam doporučené literatury:

1. FABIAN, Richard. Data-Oriented Design: Software engineering for limited resources and short schedules [online]. Richard Fabian, 2018, 307 s. ISBN 9781916478701.
2. STROUSTRUP, Bjarne. The C++ Programming Language. 4th Edition. Addison-Wesley Professional, 2013, 1376 s. ISBN 0275967301.
3. NESTERUK, Dmitri. Design Patterns in Modern C++: Reusable Approaches for Object-Oriented Software Design. New York: APress, 2018. ISBN 978-1484236024.
4. KUSSWURM, Daniel. Modern X86 Assembly Language Programming: 32-bit, 64-bit, SSE, and AVX. Apress, 700 s. ISBN 1484200659.
5. BRYANT, Randal a David O'HALLARON. Computer Systems: A Programmer's Perspective. 3rd Edition. Pearson, 1128 s. ISBN 013409266X.

Vedoucí diplomové práce: **Ing. Peter Janků, Ph.D.**
Ústav informatiky a umělé inteligence

Datum zadání diplomové práce: **2. prosince 2022**

Termín odevzdání diplomové práce: **26. května 2023**



doc. Ing. Jiří Vojtěšek, Ph.D. v.r.
děkan

prof. Mgr. Roman Jašek, Ph.D., DBA v.r.
ředitel ústavu

Ve Zlíně dne 7. prosince 2022

Prohlašuji, že

- beru na vědomí, že odevzdáním diplomové práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že diplomová práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně;
- byl/a jsem seznámen/a s tím, že na moji diplomovou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – diplomovou práci nebo poskytnout licenci k jejímu využití jen připouští-li tak licenční smlouva uzavřená mezi mnou a Univerzitou Tomáše Bati ve Zlíně s tím, že vyrovnání případného přiměřeného příspěvku na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše) bude rovněž předmětem této licenční smlouvy;
- beru na vědomí, že pokud bylo k vypracování diplomové práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky diplomové práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem diplomové práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na diplomové práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně, dne TODO

Tomáš Janečka, v.r.

ABSTRAKT

Diplomová práce se bude zabývat tématem datově orientovaného vývoje software. V teoretické části se student bude zabývat popisem jednotlivých technik návrhu software, charakteristikou používaného hardware a popisem vybraných aspektů ovlivňujících výkonnost implementovaných algoritmů. V praktické části budou srovnány jednotlivé způsoby implementace algoritmů včetně praktických příkladů doplněné o výstupy nástrojů pro výkonnostní testy a profilování.

Klíčová slova: datová orientace, C++, optimalizace, výkon, benchmarking, profilování, vyrovnávací paměť, operační paměť

ABSTRACT

This diploma thesis is concerned with the topic of data-oriented software design. In the theoretical part, the student is tasked with describing software design techniques, the characteristics of the hardware used and pointing out selected aspects that determine the performance of the implemented algorithms. The practical part shows the comparison of the different ways of algorithm implementations, along with examples including outputs from performance measuring tools and profilers.

Keywords: data-oriented, C++, optimization, performance, benchmarking, profiling, cache, memory

TODO

Prohlašuji, že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

OBSAH

ÚVOD.....	8
I TEORETICKÁ ČÁST.....	9
1 NADPIS	ERROR! BOOKMARK NOT DEFINED.
1.1 PODNADPIS.....	ERROR! BOOKMARK NOT DEFINED.
1.2 PODNADPIS.....	ERROR! BOOKMARK NOT DEFINED.
1.2.1 Podpodnadpis	13
2 NADPIS	ERROR! BOOKMARK NOT DEFINED.
2.1 PODNADPIS.....	ERROR! BOOKMARK NOT DEFINED.
2.1.1 Podpodnadpis	Error! Bookmark not defined.
II PRAKTICKÁ ČÁST.....	57
3 NADPIS	ERROR! BOOKMARK NOT DEFINED.
3.1 PODNADPIS.....	ERROR! BOOKMARK NOT DEFINED.
3.2 PODNADPIS.....	ERROR! BOOKMARK NOT DEFINED.
4 NADPIS	ERROR! BOOKMARK NOT DEFINED.
4.1 PODNADPIS.....	ERROR! BOOKMARK NOT DEFINED.
ZÁVĚR	58
SEZNAM POUŽITÉ LITERATURY.....	71
SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK.....	74
SEZNAM OBRÁZKŮ	75
SEZNAM TABULEK.....	76
SEZNAM PŘÍLOH.....	78

TODO – obsah poskládat manuálně

ÚVOD

Objektově orientované programování je jedno z nejrozšířenějších paradigmat mezi programátory. Mnoho z nich, včetně mě, se s ním setkali na úplném začátku své programátorské kariéry. Jedná se o velmi užitečný nástroj. Každý nástroj má ale svůj účel, a není určen k řešení každého problému. Jedním ze zásadních rozdílů mezi objektově orientovaným a datově orientovaným návrhem je ta věc, kterou tyto způsoby programování považují jako hlavní. Objektově orientovaný návrh se soustředí na vytvoření abstraktního, idealizovaného a také co nejobecnějšího modelu reálného problému. Naproti tomu datově orientovaný návrh považuje data za to nejvýznamnější. Přeci jenom, programy ve své podstatě slouží k transformaci dat z jedné podoby do druhé. [TODO acton2014] Zároveň se toto paradigma soustředí na charakteristiky hardware, na kterém náš software běží a dbá na efektivní využívání zdrojů. Z tohoto důvodu je tento způsob tvorby programů mimo jiné využíván v herním průmyslu. Právě herní vývojáři tvoří naučné podklady o tomto tématu, ve kterých často ukazují na podstatné nuance při souhře hardware a software. Tyto zdánlivé detaily však často mají zásadní vliv na rychlost běhu programu a také na využití operační paměti. Osobně jejich nápady považuji za velmi zajímavé, a proto by tato diplomová práce měla být shrnutím nejzásadnějších myšlenek tohoto paradigmatu. V teoretické části si představíme definici DOP a popíšeme hlavní myšlenky. Rovněž je třeba se zabývat tématem mikroarchitektury počítače, jelikož jeho znalost je pro využití v této oblasti kritická. V praktické části bude představeno množství praktik, které lze aplikovat na tvorbu programu a budou porovnána běžná řešení často řešených problémů a také zhodnoceny výstupy výkonnostních testů a profilovacích nástrojů.

TODO

I. TEORETICKÁ ČÁST

1 DATOVĚ ORIENTOVANÉ PROGRAMOVÁNÍ

V této sekci si přiblížíme pojem datově orientované programování a datově orientovaný návrh. Kromě dále uvedených definic je možné o tomto paradigmatu říci, že se jedná o způsob, jakým vyvíjet software. Zároveň ale může koexistovat s kódem, který byl napsán způsobem jiným ve stejném projektu. Tento nástroj se neváže ke konkrétní oblasti problémů či programovacích jazyků. V žádném případě se nejedná o něco, co by nebylo použito v minulosti, byť například pod jiným jménem. Ačkoliv nejde o nový koncept, samotný pojem „data-oriented“ se ve vývojářských kruzích začal vyskytovat teprve nedávno. I z tohoto důvodu je třeba při této diplomové práci využít omezené množství knih, které se zabývají tímto tématem, ovšem také větší množství záznamů přednášek z programátorských konferencí. Navíc si může člověk při studiu této problematiky všimnout, že každý řečník či autor si pod tímto pojmem představuje něco trochu jiného. Některé koncepty může zcela zanedbat a také může představit něco, o čem nikdo před ním nemluvil.

1.1 Definice

„Datově orientovaný návrh je dovednost navrhnout software pomocí vývoje transformací pro data v řádné formě, kde řádná forma je řízena cílovým hardwarem a transformacemi, které na něm běží.“ [TODO dodmain]

„Datově orientovaný návrh si nechává napovědět daty, která jsou pozorovatelná nebo očekávaná. Na rozdíl od uvažování všech možných scénářů nebo plánování adaptability, využijeme nejpravděpodobnější vstupy pro nasměrování algoritmu. Na rozdíl od plánování rozšiřitelnosti je jednoduchý a má za cíl splnit svůj úkol.“ [TODO dodmain]

1.2 Historický výskyt

Článek na téma „data-oriented“, který jako jeden z prvních použil tento termín a také měl za cíl seznámit čtenáře s touto myšlenkou, vyšel v roce 2009 v časopisu Game Developer. Jedná se o příspěvek od herního vývojáře o způsobu vývoje her v časopisu pro herní vývojáře. Není divu, že toto paradigma pramení právě z oblasti, kde je souhra software a hardware klíčem k úspěchu. V tomto článku autor pojednává o tom, jak by všudypřítomné objektově orientované programování mohlo být příčinou nízkého výkonu her. Je v něm uvedeno, na kterou věc se různé programovací přístupy soustředí a jak se od nich datově orientovaný přístup liší. Autor dále uvádí svůj výčet výhod tohoto přístupu, a to paralelizace, využití vyrovnávací paměti, modularita a jednoduchost testování. Závěrem jsou představeny rady,

jak začlenit tento přístup do aktuálně vyvíjené aplikace a jak data získat a co je důležité sledovat. Autor se ještě vyjadřuje k tomu, že pro objektově orientovaný návrh rozhodně existuje místo a nechce ho demonizovat. Například „v systémech, které byly navrženy tímto způsobem nebo výkonově nekritické aplikace.“ [TODO llopiš] Odkazy z tohoto příspěvku míří na Mika Actona a Jima Tilandera. Oba jsou vlastníky webů, kde v minulosti publikovali blogy na různá témata, které zjevně ovlivnily Noelův přístup k vývoji aplikací a her. Z toho je zřejmé, že myšlenky datově orientovaného návrhu pramení z prvních let druhého tisíciletí.

Jelikož se zde také klade důraz na vývoj software s ohledem na hardware, dalo by se říct, že byl tento přístup používán ještě mnohem dříve. V raných dobách výpočetní techniky byly, v porovnání s dnešní dobou, všechny dostupné prostředky velmi vzácné, a proto bylo nezbytné s nimi nakládat co nejefektivněji.

1.3 Hlavní myšlenky

1.3.1 Je to o datech

„Data jsou vše, co máme“. Všechny aplikace, co kdy byly napsány, slouží k poskytnutí výstupu v závislosti na vstupních datech. Grafické aplikace pracují s obrázky. Textové editory pracují s textem. Každá z nich očekává určitý formát dat. Ten může být velmi složitý, nebo velmi jednoduchý. Programátoři si také často neuvědomují, že instrukce jsou také data, protože se rovněž nachází v operační paměti. Za všech okolností je potřeba myslet na to, že data nikdy neexistují jen tak v éteru, ale pokaždé se nachází na nějakém hardware, ať už na virtuálním stroji, nebo konkrétním procesoru. [TODO dodmain]

1.3.2 Data nejsou problémová doména

„Datově orientovaný návrh nezabudovává problém z reálného světa do kódu“. Na rozdíl od objektově orientovaného přístupu, datově orientovaný přístup obětovává čitelnost kódu, což nám umožňuje nezatěžovat počítač lidskými koncepty. Umístěním dat do třídy nám umožní dát těmto datům kontext, to ale může mít následek existence velkého množství dat, které spolu nesouvisí. Proto v tomto paradigmatu uvažujeme o datech jako o „faktech, o kterých můžeme uvažovat tak, jak potřebujeme pro získání výstupních dat v požadovaném formátu.“ [TODO dodmain]

1.3.3 Statistika

„Data jsou typ, frekvence, množství, tvar a pravděpodobnost.“ [TODO dodmain]

Nejen vstupní data programu jsou zahrnuta do pojmu „data“. Data o datech mohou být stejně nebo i více významná. Mohou mít zásadní vliv na to, jak píšeme kód. Pokud máme k dispozici prvotní verzi funkční aplikace, která pracuje s produkčními daty, nebo disponujeme daty, která byla shromážděna libovolným způsobem, máme nyní více kontextu a dokážeme lépe uvažovat o problému, který řešíme. „Pokud porozumíme datům, porozumíme problému.“ Analýza dat může mít podobu prostého výpisu hodnoty proměnné. Stačí, když zvolíme libovolnou proměnnou, která nás zajímá a budeme sledovat vývoj jejích hodnot v čase. [TODO acton2014] Jelikož vývoj aplikací může být velmi náročný na čas a prostředky, určitě je rozumné investovat naše úsilí do 20 % kódu, ve kterém je tráveno 80 % času a má zásadní vliv na výkon programu.

1.3.4 Data se mění

„Datově orientovaný návrh může pozorovat změnu v architektuře aplikace porozuměním změnám v datech.“ Významná myšlenka je držet data a operace nad těmito daty zvlášť a neshlukovat je do tříd nebo jiných logických konstruktů. Díky tomu dokážeme lépe reagovat na změny a minimalizovat náklady potřebného přepisu kódu. [TODO dodmain]

1.4 Využití

Jak již bylo zmíněno, datově orientovaný návrh se hojně využívá v herním průmyslu. Je to jedna z oblastí, kde se vývojáři snaží vytěžit co možná největší výkon ze své aplikace a zároveň musí respektovat omezení jednoho nebo více druhů hardware, na kterém bude běžet. Po seznámení s tímto paradigmatem a jeho hlavními myšlenkami a způsoby implementace je mi zřejmé, že použití tohoto přístupu má pro programátory jako jednotlivce i další zajímavé implikace. Jedna z nich je zařazení dalšího užitečného nástroje mezi své dovednosti. Jelikož pro aplikování datově orientovaného návrhu je důležitá znalost hardware, je programátor nucen se vzdělávat v oblasti počítačové architektury, mikroarchitektury, operačních systémů a strojového kódu. Zároveň je zde potenciál lepšího porozumění problému, který je zrovna řešen programátorem. Protože pokud porozumíme datům, porozumíme problému, může člověk zjistit řadu hodnotných informací a podle toho může v budoucnu vylepšit kód. Mezi specifické informace by se dala zařadit frekvence volání určitých funkcí, studium hodnot proměnných měnících se v čase nebo vypočítávání vzorce opakování hodnot proměnných.

1.4.1 Podpodnadpis



Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Obrázek 1. Ukázkový obrázek

Tabulka 1. Ukázková tabulka

2 DALŠÍ PARADIGMATA

Programovací paradigma je způsob nebo styl, kterým píšeme kód. Nemusí se nutně vztahovat ke konkrétnímu programovacímu jazyku, i když různé programovací jazyky mají blíže k jednomu paradigmatu než ke druhému. V následující sekci se nachází srovnání paradigmat, kterým se zabývá tato diplomová práce, s dalšími paradigmaty, které se běžně používají, nebo začínají používat v aktuální době. Byla vybrána pouze ta, která jsou pro srovnání relevantní, protože mají například řešit nedostatky druhého.

2.1 Objektově orientované programování

„Objektově orientované programování je o modelování systému jako kolekci objektů, kde každý objekt představuje určitý aspekt systému. Objekty obsahují jak funkce, tak data. Objekt poskytuje veřejné rozhraní, které je přístupné v kódu, a také obsahuje svůj privátní, vnitřní stav; ostatní části systému se nemusí zajímat o to, co se děje uvnitř objektu.“ [TODO mozillaOOP]

V rámci tohoto paradigmatu se hojně využívá tříd. Třída je předpis pro vytváření instancí. Každá třída může mít vlastnosti, které charakterizují instance, a metody, které popisují chování. Mezi třídami mohou existovat vztahy. Jedním z nich je dědičnost, díky které jedna třída dědí vlastnosti a metody třídy druhé. [TODO mozillaOOP] Pokud rodičovská třída definuje metody jako virtuální, pak třídy, které od ní dědí, mohou měnit chování v závislosti na typu instance. Toto nám umožňuje mít kolekci objektů různého typu, ale pro interakci s nimi používat jednotný přístup. Ačkoliv se jedná o užitečnou věc, jsou to právě volání virtuálních funkcí, které v určitých případech mohou být příčinou zásadního zpomalení aplikace.

Zapouzdření je myšlenka vymezení veřejného rozhraní a zároveň schování detailů fungování objektu. V případě potřebné změny nám toto umožní změnit kód pouze na jednom místě, jelikož navenek je přístupné pouze veřejné rozhraní, které nebylo třeba měnit. [TODO mozillaOOP]

2.2 Funkcionální programování

„Funkcionální programování je přístup k vývoji software, který používá ryzí funkce pro vytvoření udržitelného software. Jinými slovy se jedná o tvoření programů aplikací a kompozicí funkcí.“ [TODO wtfp]

Jak vyplývá z názvu, funkce je zde základní stavební jednotka. Kromě jejich běžného použití jsou funkce využívány taky jako proměnné, argumenty funkcí nebo návratové hodnoty funkcí. Na rozdíl od ostatních paradigmat se zde preferuje použití proměnných, jejichž hodnota se po deklaraci nemění. Základní myšlenky tohoto programovacího stylu pochází z matematického nástroje zvaného lambda kalkul, který byl popsán ve 30. letech minulého století Alonzo Churchem. Mezi funkcionální programovací jazyky se řadí Haskell, Erlang, Clojure, LISP, Scala a Elixir. Postupem času i běžné programovací jazyky zařazují do svého arzenálu nástroje, které pramení z tohoto způsobu programování. [TODO wtfp]

Ve funkcionálním programování je významné používání rekurze. V závislosti na implementaci toto může způsobit značné zpomalení aplikace v porovnání s implementací pomocí použití klasické iterace ve smyčce.

2.3 Porovnání

„Objektově orientovaný návrh je soustředěn na problém a jeho řešení. Objekty, abstraktní reprezentace věcí, které tvoří návrh řešení problému představeného v návrhovém dokumentu aplikace. Objekty manipulují pouze s těmi daty, které jsou potřeba pro jejich reprezentaci bez jakéhokoliv ohledu na hardware nebo na data z reálného světa nebo jejich množství. Z tohoto důvodu nám objektově orientovaný návrh umožní rychle sestavit první verze aplikací a tím pádem také první podobu kódu. Datově orientovaný návrh se k problému staví jinak. Na rozdíl od předpokladu, že nevíme nic o hardware, usuzujeme, že nevíme nic o řešení problému.“ [TODO dodmain]

Rozdíl mezi objektově orientovaným a funkcionálním přístupem je ten, že zatímco OOP využívá imperativní přístup, který spočívá ve specifikaci kroků potřebných k vyřešení problému, FP využívá deklarativní přístup, který pracuje s výsledkem operace, nehlédě na to, jak jsme k němu přišli. Dalším rozdílem je využití proměnných a konstantních proměnných. V FP se v případě přepisu vytvoří zcela nová proměnná, do které se přepíše původní hodnota. Díky tomu se kód v případě potřeby snáze mění a lépe testuje a lépe se v něm hledají chyby. Autor doporučuje využívat OOP pro standardizované a přímočaré projekty a FP pro aplikace, které je třeba škálovat a musí být flexibilní. [TODO wtfp]

Jedna věc, kterou sdílí funkcionální programování s datově orientovaným programováním je skutečnost, že obě paradigmata identifikovala nedostatky objektově orientovaného programování. Jejich společná překážka je potřeba uvažovat zcela odlišně, než jak člověk

doposud myslel při psaní programu. Pro programátora odchovaného na objektově orientovaném návrhu mohou být myšlenky aplikované v těchto alternativních programovacích přístupech relativně složité, ba i zpočátku nepochopitelné. Obě paradigmaty se mohou vyskytovat po boku objektově orientovaného kódu ve stejné aplikaci a být použity tam, kde dávají smysl. Zatímco funkcionální návrh si klade za cíl zlepšit robustnost a modularitu aplikace, datově orientovaný přístup se soustředí jak na modularitu, tak na zvýšení výkonu programu.

3 HARDWARE JAKO PLATFORMA

Datově orientovaný vývoj dbá na efektivní využívání hardware. V této sekci si popíšeme části počítače a jednotlivých komponentů, které mají zásadní vliv na rychlost běhu programu.

3.1 Vyrovnávací paměť

Vyrovnávací paměť (cache) je rychlé paměťové zařízení s malou kapacitou, které slouží k ukládání malých částí dat z paměťových zařízení nižších úrovní paměťové hierarchie. Pokud se bavíme o cache pamětech, máme zpravidla namysli uložistiště, které se nachází na procesorovém čipu a je k dispozici výpočetním jádrům. Jako vyrovnávací paměť ale můžeme považovat i operační paměť ve vztahu k hard-disku nebo SSD. [TODO csprogrammer]

Cache paměť pro ukládání dat využívá technologie SRAM. V moderních procesorech se vyskytuje v několika úrovních. Každá úroveň má různou velikost a přístupovou dobu. Nejblíže k výpočetnímu jádru je úroveň L1. Tu ještě výrobci CPU separují na paměť pro instrukce a data, nazývané *i-cache* a *d-cache*. Tato úroveň bývá privátní pro jedno jádro. Na další úrovni se nachází úroveň L2. Ta je unifikovaná, takže obsahuje jak instrukce, tak data. Může být privátní pro jedno jádro nebo sdílená mezi všemi jádry. Nejvýše postavená je úroveň L3. Ta je společná pro všechna jádra a má největší kapacitu. [TODO csprogrammer]

Vývoj vyrovnávacích pamětí je jeden z přispěvatelů k relativně obrovským výkonům dnešních CPU. Tento růst bohužel mnohonásobně převyšuje ten u operačních pamětí. I z tohoto důvodu by se měli programátoři naučit, jak efektivně využívat cache paměť. [TODO caqa]

Bity s slouží pro nalezení odpovídající sady v cache paměti. Bity t slouží pro identifikaci odpovídající cache line v sadě. Bity b nám říkají, kde v rámci je bloku začátek dat, o která žádáme. [TODO csprogrammer]

3.1.2 Pojmy

Při popisování funkce vyrovnávacích pamětí je třeba si popsat několik běžně používaných pojmů. *Cache hit* označuje nalezení požadovaných dat v první sousední úrovni směrem dolů v paměťové hierarchii. Při optimalizaci programů je snaha uchovat data, se kterými se operuje, v jakékoliv úrovni cache na procesoru, jelikož i přístup do L3 je rychlejší než přístup do operační paměti. *Cache miss* označuje absenci požadovaných dat v první sousední úrovni směrem dolů v paměťové hierarchii. Je nutné, aby byla data načtena z úložiště a uložena do cache. Pokud je cache paměť plná, je třeba využít zvolené substituční strategie a nahradit nějakou cache line. Toto může být označeno jako *block eviction*. Existuje několik druhů *cache miss*. V případě, že je vyrovnávací paměť prázdná, při žádosti o data nastává *compulsory miss*. Pokud by velikost vyrovnávací paměti byla příliš malá, nebo bychom se do ní pokoušeli zapisovat data, která musí být uložena na stejné místo, jedná se o *conflict miss*. Jestliže pracujeme obrovskou sadou dat, iterujeme přes ni ve smyčce a tím dojde k vyčerpání kapacity cache, pozorujeme *capacity miss*. Když dochází k opakovaným konfliktům na stejném místě v cache paměti, označujeme to jako *thrashing*. Poměr počtu *cache miss* a počtu dotazů na data nám dává *miss rate*. *Hit rate* je vyjádřeno jako $1 - \text{Miss rate}$. Doba přesunu dat z paměti do CPU se označuje jako *hit time*. Doba čekání na data v případě *cache miss* se jmenuje *miss penalty*. Bavíme-li se o pojmu *write hit*, máme na mysli to, že se data, která chceme aktualizovat, vyskytují v nejbližší cache paměti. Naproti tomu *write miss* označuje absenci dat, která aktualizujeme. [TODO csprogrammer]

3.1.3 Vyrovnávací paměť s přímým mapováním

Direct-mapped cache je typ vyrovnávací paměti, který má v každé sadě právě jednu cache line. Tyto paměti jsou jednoduché na implementaci a používání, ale je zde zvýšené riziko *thrashingu*. [TODO csprogrammer]

3.1.4 Set associative vyrovnávací paměť

Tento typ pamětí se redukuje problém existující v paměti s přímým mapováním pomocí navýšení počtu cache line v rámci jedné sady. *Asociativní paměť* ukládá data jako pole dvojic

klíč-hodnota. Každou sadu v této paměti si lze představit jako asociativní paměť, jejímž klíčem je spojení bitů v a t a hodnotou je obsah bloku. [TODO csprogrammer]

Při návrhu stupně asociativity je třeba zvolit vhodný kompromis mezi větším a nižším. Vyšší stupeň asociativity je pomalejší a složitější na implementaci. [TODO csprogrammer]

Cache		
L1 Data	4 x 32 KBytes	8-way
L1 Inst.	4 x 32 KBytes	8-way
Level 2	4 x 256 KBytes	4-way
Level 3	6 MBytes	12-way

Obrázek 4: Příklad uspořádání a typu cache paměti – snímek programu CPU-Z.

Zdroj vlastní.

Zde je příklad různých úrovní cache paměti. V levém sloupci vidíme *počet jader x velikost paměti*. V pravém sloupci je uveden stupeň asociativity. *N-way* nám říká, že každá sada pojme N cache line.

3.1.5 Plně asociativní vyrovnávací paměť

Jedná se o asociativní vyrovnávací paměť s jednou sadou. Tato sada obsahuje všechny cache line. Při adresování je možné vypustit bity s , jelikož se vždy pracuje s hodnotou 0. Plně asociativní vyrovnávací paměť je vhodná pro malé kapacity, jelikož pro větší kapacity by bylo zapotřebí značné množství hardware a zároveň by byla pomalá. [TODO csprogrammer]

3.1.6 Substituční strategie

Pokud nastane *conflict miss*, je třeba umístit žádaná data na vhodné místo. Toto místo může být zvoleno náhodně. Alternativně můžeme zvolit sofistikovanější postupu, jako je nahrazení dat, která byla použita nejdále v minulosti (LRU), případně nejméně často využita ve stanoveném časovém okně (LFU). [TODO csprogrammer]

3.1.7 Způsoby zapisování

Na rozdíl od čtení dat, zápis dat je o něco složitější. Jsou-li data, která aktualizujeme, v cache paměti, můžeme buď nová data rovnou zapsat do cache paměti nižší úrovně (*write-through*) nebo upravit data a zapsat je do paměti až v době vyřazení cache line z paměti (*write-back*). Nastane-li *write miss*, máme opět na výběr. Způsob *write-allocate* si nejdříve načte blok z nižší paměťové vrstvy a poté upraví data. Na rozdíl od toho, *no-write-allocate* přímo zapíše

data do nižší úrovně. Výše uvedené způsoby lze kombinovat mezi sebou a každá kombinace je vhodná pro jiný cíl. [TODO csprogrammer]

3.2 Operační paměť

Operační paměť na rozdíl od vyrovnávacích pamětí využívá technologie DRAM pro ukládání dat. Jeden bit je uchován pomocí kondenzátoru. Kvůli pokročilému stupni integrace a zároveň náchylnosti na rušení, každý kondenzátor musí být pravidelně dobíjen (refresh). Tento způsob ukládání dat je využit pro operační paměti kvůli možnosti dosažení vyšší kapacity na stejnou plochu čipu v porovnání s SRAM a také kvůli ceně. Čas přístupu je ale delší. [TODO csprogrammer]

Data na paměťovém čipu jsou uspořádána do dvourozměrného pole označovaného jako superbuňka (supercell) a ta zase obsahuje určitý počet paměťových buněk (cell). Čip je propojen s paměťovým kontrolérem pomocí adresových a datových vodičů, které slouží k vyhledání požadovaných buněk a zápisu nebo čtení dat. Při adresování se nejprve na vodiče zapíše adresa řádku, který se následně celý zkopíruje do interního bufferu řádku, a následně se na stejné vodiče zapíše adresa sloupce, což zapříčiní zápis dat z požadované paměťové lokace z bufferu řádku na datové vodiče. [TODO csprogrammer]

Paměťové čipy jsou dále uspořádány do paměťových modulů. Přístup jednotlivým čipům na modulu probíhá paralelně. V případě čtení dat z paměti paměťový kontrolér obdrží adresu, kterou rozdělí podle počtu čipů na modulu pro získání adresy superbuňky. Každý čip zapíše hledaná data zpět na výstup modulu, který data poskládá do správného pořadí a poté je pošle kontroléru. Data jsou pomocí sběrnice následně zaslána do CPU. [TODO csprogrammer]

3.3 CPU pipelining

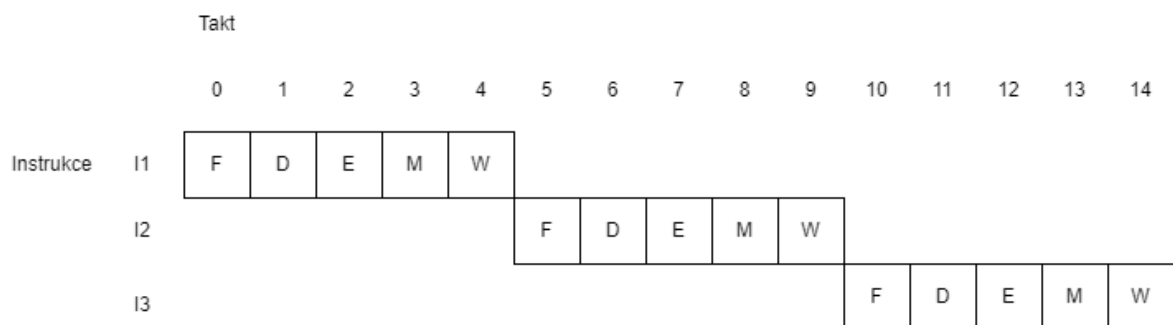
3.3.1 Fáze instrukčního kanálu

Zpracování jedné instrukce procesorem je rozděleno do několika fází a bývá popsáno instrukční *pipeline*. První je fáze *fetch*, ve které se načte instrukce z paměti pomocí adresy, která je uchována v registru jménem *program counter*. Následující úroveň *decode* se zabývá načtením operandů. Krok *execute* provede operaci, kterou popisuje instrukce, pomocí aritmeticko-logické jednotky. Může se jednat o matematické nebo logické operace, výpočet adresy, vyhodnocení podmínky nebo směr větvení. Zápis a čtení paměti se děje ve fázi *memory*. Konečný krok *write-back* zapisuje vypočtené výsledky do registrů. Abychom mohli hodnotit

vhodnost různých přístupů k provádění instrukcí, používáme pojmy *latency*, což je doba vykonání operace, a *throughput*, který popisuje počet provedených operací za jednotku času. [TODO csprogrammer]

3.3.2 Sekvenční zpracování

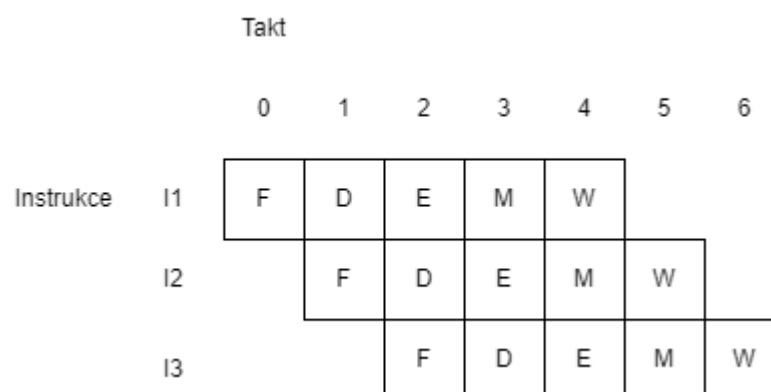
Jestliže vykonáváme instrukce jednu po druhé a každá instrukce projde kanálem jako celek, hovoříme o sekvenčním vykonávání.



Obrázek 5: Sekvenční vykonávání instrukcí. Zdroj vlastní.

3.3.3 Skalární pipeline

Jiný způsob provádění instrukcí, který by měl lépe využívat dostupný hardware, se nazývá *pipelined*.



Obrázek 6: Skalární instrukční pipeline. Zdroj vlastní.

3.3.4 Superskalární pipeline

Vylepšení skalárního *pipelined* přístupu je způsob superskalární. Ten je zdokonalen pomocí přidaného hardware, díky kterému se může během jednoho taktu nacházet více instrukcí v té samé fázi.

		Takt						
		0	1	2	3	4	5	6
Instrukce	I1	F1	D1	E1	M1	W1		
	I2	F2	D2	E2	M2	W2		
	I3		F1	D1	E1	M1	W1	
	I4		F2	D2	E2	M2	W2	
	I5			F1	D1	E1	M1	W1
	I6			F2	D2	E2	M2	W2

Obrázek 7: Superskalární instrukční pipeline. Zdroj vlastní.

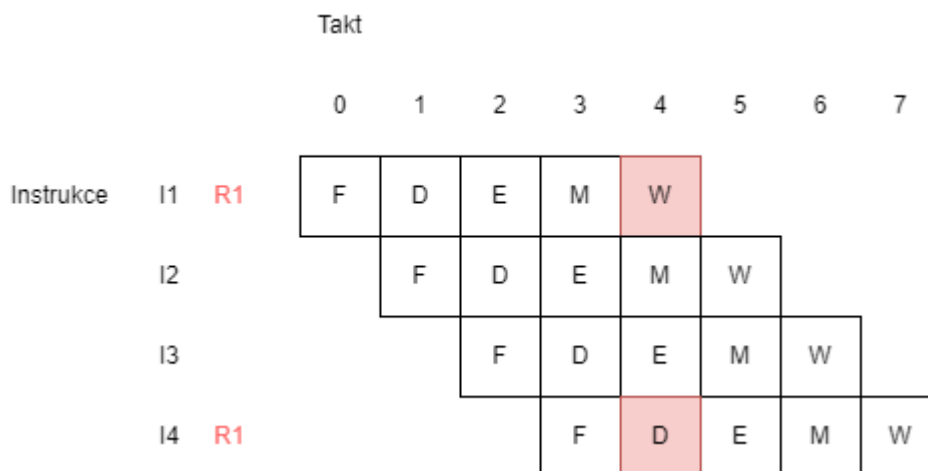
3.3.5 Ideální vs reálné řešení

Ačkoliv je *pipelining* vhodným způsobem pro zvýšení počtu vykonaných instrukcí za jednotku času, může mít za následek mírný nárůst latence každé instrukce. Ve dříve popisovaných příkladech *pipeline* jsou uvažovány uniformní časy strávené v každé fázi. Ve skutečnosti však mohou různé kroky trvat různě dlouho a tím pádem způsobovat prodlevy při přechodu mezi kroky. Toto může být částečně vyřešeno zvýšením počtu fází a snížením doby trvání každé fáze, což, jako každé rozhodnutí při návrhu hardware, má taky své nevýhody. [TODO csprogrammer]

3.3.6 Závislosti

Doposud byla uvažována nezávislost dvou po sobě jdoucích instrukcí. V reálném světě k takovému ideálnímu případu nemusí docházet často. Rozlišujeme *datovou závislost*, kde se dvě instrukce odkazují na to samé místo v paměti nebo ten samý registr, a *kontrolní závislost*, ke které dochází při skoku nebo návratu z rutiny. Nesprávná implementace řešení těchto

stavů může zapříčinit chyby ve výpočet, které bývají označovány jako *data* a *control hazard*.
[TODO csprogrammer]



Obrázek 8: Data hazard. Zdroj vlastní.

Data hazard může mít podobu načtení staré hodnoty aktuální instrukcí, zatímco předchozí instrukce už měla hodnotu změnit na novou. Obrázek 7 ukazuje vznik chyby ve fázi D u instrukce I4. Jelikož je hodnota R1 aktualizována ve stejném taktu, jako je načítání hodnot operandů instrukce, načtená hodnota není správná. Tento problém lze vyřešit vícero způsoby. Ten nejjednodušší je pozastavení (stalling) pipeline. Spočívá ve vložení takzvaných bublin (bubble) mezi problémové instrukce, aby se posunuly překrývající fáze v rámci cyklu a tím pádem byla načtena správná hodnota. Tento přístup může mít negativní vliv na počet provedených instrukcí za jednotku času. Další způsob se nazývá *data forwarding*. Namísto toho, abychom nová data zapisovali ve fázi W do souboru registrů, aby mohla být v dalším taktu načtena fází D, můžeme aktualizovanou hodnotu předat přímo z W na požadované místo. [TODO csprogrammer]

Pokud ve fázi F nedokážeme jednoznačně určit adresu další instrukce, mluvíme o *control hazard*. Příkladně u instrukce RET, po jejímž zavolání by se obecně měl vrátit proud vykonávání na místo předchozího volání funkce, je adresa další instrukce známá až ve fázi W a efektivně tedy dochází k pozastavení pipeline. V případě instrukce podmíněného skoku se nemusí čekat na výsledek porovnání, ale lze odhadnout, kterým směrem větve se bude pokračovat. Jen tehdy, pokud se nesprávně zvolená instrukce nachází ve fázích F nebo D je můžeme po detekci bez problémů odstranit z pipeline, protože ještě neměly čas změnit aktuální kontext. [TODO csprogrammer]

Dalším příkladem je *structure hazard*, který je zapříčiněn omezeným počtem exekučních jednotek. Může nastat, jestliže dvě navzájem se překrývající instrukce bojují o tu samou funkční jednotku. Předchozí obrázek zobrazuje možný *structure hazard* v taktu 3, kdy se překrývá fáze F s fází M. V případě, že by obě tyto fáze využívaly stejný paměťový modul a ten podporoval pouze jednotlivé přístupy, nastává *structure hazard*. Tento případ by se dal opět vyřešit vložením bubliny. [TODO caqa]

3.4 Prefetching

Prefetching je jeden ze způsobů tolerance latence načítání dat z paměti. Zabývá se načítáním dat z paměti předtím, než jsou potřeba, například pomocí predikce adres. Tímto způsobem můžeme eliminovat *compulsory cache miss*. V moderních systémech se velikost dat, která jsou přednačtena, rovná velikosti cache bloku. Tato technika může být provedena hardwarově, na úrovni kompilátoru nebo uživatelsky. Existuje více druhů prefetcherů a ty se mohou lišit tím, který přednačítací algoritmus je použit. Přednačítací algoritmus udává, co se bude načítat. Většina moderních systémů ukládá přednačtená data do cache paměti. [TODO mutluPrefetch]

3.4.1 Softwarový prefetching

Prefetching pomocí software je proveden instrukcemi pro přednačítání dat, které poskytuje instrukční architektura. Pracuje s nimi programátor nebo kompilátor a jsou vhodné pro přístupy do paměti, které jsou pravidelné. Je možné vybrat si, do které úrovně cache paměti budou data zapsána. [TODO mutluPrefetch]

3.4.2 Hardwarový prefetching

Prefetching pomocí hardware spočívá v monitorování přístupů do paměti a nalezení opakujících se vzorů. Předpovězené adresy jsou generovány automaticky. Na rozdíl od softwarového přístupu, hardwarový způsob může být lépe přizpůsoben systému, který používáme, a také může představovat menší zatížení. Funkce takového prefetcheru může být taková, že při čtení určitého cache bloku z paměti automaticky načte i jeden nebo více dalších, které po něm následují (*next-line prefetcher*). Jestliže je zaznamenáno několik opakujících se instrukcí, které načítají z paměti s pravidelným offsetem od počáteční adresy, může v určitém případě dojít k přednačtení dalších pravděpodobných dat (*stride prefetcher*). [TODO mutluPrefetch]

3.4.3 Execution based prefetching

Univerzální způsob, který může být implementován jak hardwarově, tak softwarově, je *execution based prefetching*. Zde probíhá přednačítání dat v jiném vlákne, než ve kterém je prováděn hlavní program. [TODO mutluPrefetch]

Ve speciálním vlákne stačí provést ty části kódu, při kterých by jinak nastal *cache miss*. Mezi ně patří výpočet adres, předpovídání větvení a předpovídání hodnoty. Toto vlákno může existovat jak na stejném jádře, jako je vykonávaný kontext, tak na zcela separátním jádře. Rovněž nemusí existovat po celou dobu běhu programu, ale může dynamicky vznikat a zanikat. Pokud to ISA nabízí, může programátor využít poskytované instrukce pro zahájení výpočtu problémového bloku kódu. [TODO mutluPrefetch2]

3.4.4 Address correlation based prefetching

Tvrzení, že se snáze odhadují pravidelně se posouvající přístupy do paměti, není překvapující. Ne u všech datových struktur, použitých pro práci s daty, je toto možné. Proto existují i různé přístupy k prefetchingu, které se snaží zrychlit i nepravidelné přístupy do paměti. Tento způsob využívá korelační tabulku pro zaznamenání přímého přístupu do paměti, a zároveň také těch, které následovaly hned po něm. Lze totiž vypořizovat, že některé přístupy mohou nastat s větší pravděpodobností než ostatní. Toto lze znázornit Markovovým modelem. Ke každému odkazu do paměti může být zapamatován jeden nebo více dalších následujících odkazů a ty budou v budoucnu přednačteny. Tento druh prefetchingu ovšem nedokáže eliminovat *compulsory cache miss*. [TODO mutluPrefetch2]

3.4.5 Content directed prefetching

Při programování v jazycích jako C/C++ se často setkáváme s prací s ukazateli do paměti, zejména pak u některých datových struktur. Jelikož ukazatel se může odkazovat na jakékoliv místo v paměti, jen těžko bychom mohli využít předpovídání adres s pravidelným krokem. Právě pro zefektivnění práce s ukazateli řeší tento přístup. Snaží se identifikovat hodnoty v cache bloku, které jsou potencionálními ukazateli, a načíst pro ně odkazovanou hodnotu z paměti do cache. Není zde třeba žádné tabulky pro sledování přístupů v minulosti, ale také je zde možnost zahlcování cache paměti nepotřebnými daty, protože tento algoritmus načte data pro všechny ukazatele v cache bloku. [TODO mutluPrefetch2]

3.5 Lokalita

Pojem lokalita se týká odkazování na data v paměti. Je to vlastnost dobře napsaných programů, které díky práci s ní mohou běžet rychleji, a to díky faktu, že hardware počítače i operační systémy jsou navrženy pro využití lokality. Můžeme se s ní setkat v různé podobě v každé úrovni paměťové hierarchie. Zásadní myšlenka, která mohla podnítit vznik vyrovnávacích pamětí, byla, že pokud se jednou odkážeme na určité místo v paměti, je zde pravděpodobnost, že se na něj odkážeme vícekrát v blízké budoucnosti. Toto bývá označováno jako *časová lokalita*. Jiným případem je *prostorová lokalita*, která říká, že pokud se odkážeme na určité místo v paměti, nejspíše se budeme odkazovat i na další místa v paměti v blízkosti původního. [TODO csprogrammer]

3.6 Jazyk symbolických adres

V dnešní době masivního rozšíření high-level programovacích jazyků, jako je C#, Java, Python či JavaScript, se snáze zapomene na to, jak hardware počítače vlastně vykonává náš program. Je třeba si uvědomit, že pohodlnost používání těchto jazyků je výhodná zejména pro jeho uživatele. CPU počítače však netíží abstrakce, které do návrhu vnáší programátor, čímž i do určité míry zakrývá realitu toho, co se skutečně pod pokličkou děje. Skutečnost je taková, že procesor rozumí pouze konečné sadě instrukcí a umí je vykonat rychle. Nejblíže této úrovni, nebereme-li v potaz strojový kód, je jazyk symbolických adres. Porozumět kódu, který jsme napsali, v podobě jazyku symbolických adres je velmi užitečné a v kombinaci s pochopením architektury počítačového systému se jedná o velmi mocný nástroj při optimalizaci kódu.

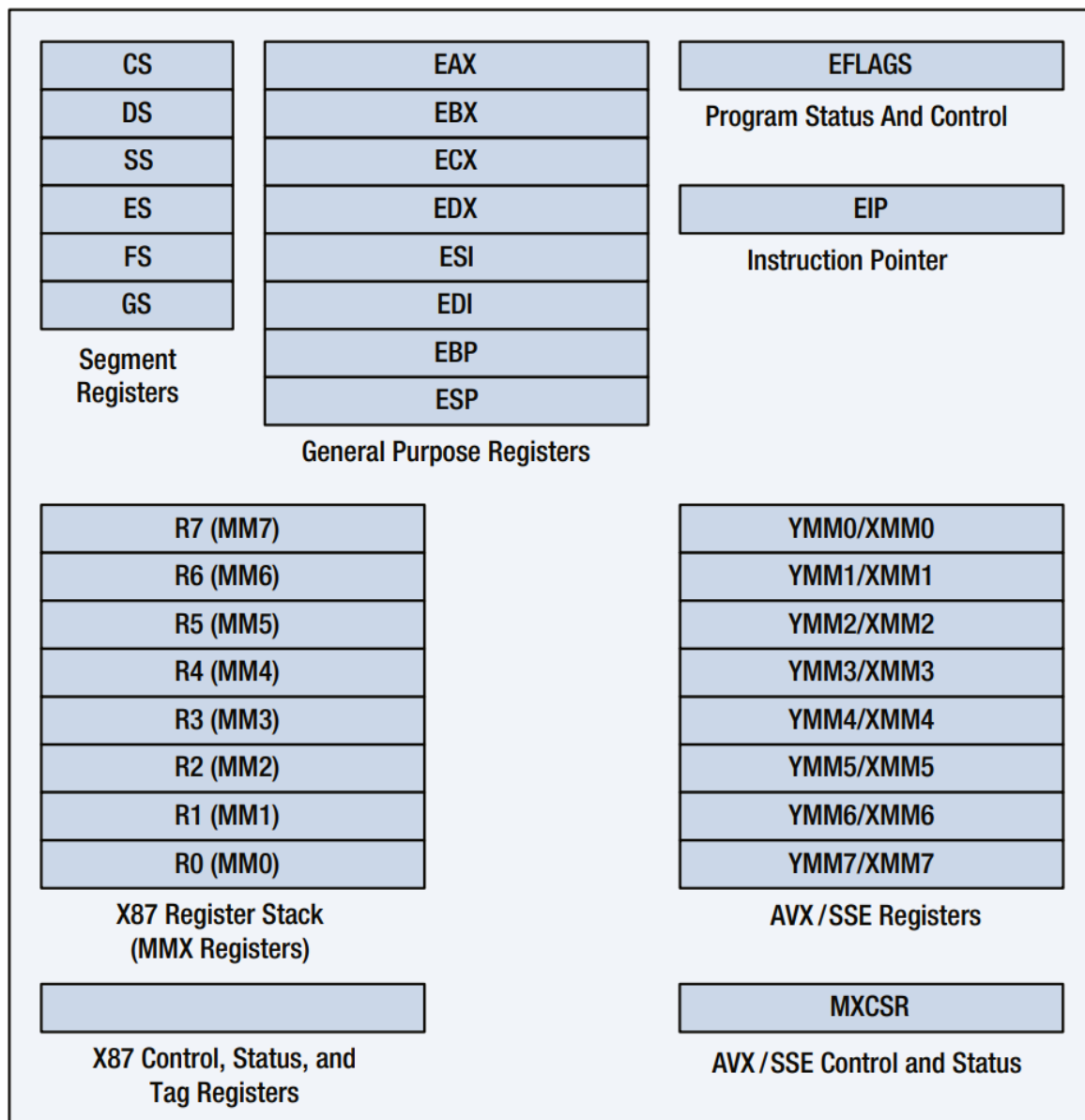
Počítače vykonávají strojový kód, což je sekvence bytů popisující operace pro manipulaci s daty, čtení a zápis dat nebo síťovou komunikaci. Z high-level jazyku vznikne strojový kód pro konkrétní stroj s použitím kompilátoru. V jazyce symbolických adres programujeme pomocí skládání instrukcí za sebe jednu po druhé. Dostupné instrukce jsou definovány abstrakcí nazývanou *instruction set architecture*. Jedny z těch nejvyužívanějších jsou x86-32 a x86-64. [TODO csprogrammer]

3.6.1 Vnitřní architektura x86-32

Název instrukční sady x86 se odkazuje na procesory společnosti Intel, které byly pojmenovány 8086, 80286, 80386 a 80486. Postupem času docházelo k modernizaci parametrů jako zvětšení velikosti registrů a dalšímu rozšiřování instrukční sady. V dnešní době jsou

definovány fundamentální datové typy jako *byte* (8 bitů), *word* (16 bitů), *doubleword* (32 bitů) či *quintword* (80 bitů). Bity každého datového typu jsou uspořádány v pořadí *little-endian*. [TODO asm]

Programátorovi je k dispozici několik sad registrů. *Segmentové registry* jsou použity pro definování logického paměťového modelu pro vykonávání programu a ukládání dat. *General-purpose registry* primárně slouží pro vykonávání logických, aritmetických a paměťových operací. Každý z nich může být dále rozdělen pro výpočty s 8 nebo 16 bitovými operandy. Ačkoliv mají sloužit k obecným účelům, v určitých případech slouží k uložení předem daných výsledků operací. Rovněž existuje konvence, která stanovuje jaký registr by se měl k čemu použít. *EFLAGS registr* obsahuje kolekci bitů převážně využívanou operačním systémem. Tyto bity indikují výsledky logických a aritmetických operací. *Instrukční ukazatel* udává, která další instrukce se bude vykonávat. Je využit zejména instrukcemi pro volání rutin, skoky nebo návraty z funkce. [TODO asm]



Obrázek 9: Vnitřní architektura x86-32. [TODO asm]

3.6.2 Adresovací módy

Popisovaná instrukční sada nabízí 4 adresovací módy. U instrukcí, které čtou nebo zapisují do registrů nebo paměti, se vždy vypočítává efektivní adresa. Efektivní adresa může být dána *bázovým registrem*, za který může být dosazen libovolný *general-purpose registr*. Dále se k adrese přičítá *displacement*, což je číselná hodnota udávající konstantní posuny zakódované v instrukci. K adrese může být dále přičtena hodnota *indexového registru*, za který může být dosazen libovolný *general-purpose registr* kromě ESP. *Indexový registr* pak ještě můžeme vynásobit *škálovacím faktorem*, který nabývá hodnot 1, 2, 4 a 8. Pro stanovení efektivní adresy můžeme použít libovolné kombinace těchto hodnot. [TODO asm]

3.6.3 Instrukce pro přesun dat

Instrukce *mov* je používána pro kopírování dat mezi registry a mezi paměťovými lokacemi. Rovněž se zde nachází instrukce podmíněného přesunu *cmovcc*, která spolupracuje s instrukcí pro porovnávání *cmp*. Instrukce *push* umístí stanovenou hodnotu na zásobník a instrukce *pop* vrátí hodnotu na vrcholu zásobníku. [TODO asm]

3.6.4 Instrukce pro aritmetické operace.

Aritmetický součet je proveden pomocí instrukce *add*, která dokáže sečíst dva operandy. Obdobně funguje instrukce *sub* pro odečítání. Instrukce pro násobení existuje ve variantě se znaménkem a bez znaménka. Verze *mul* pro násobení bez znaménka provede vynásobení dosazené hodnoty s hodnotou registru EAX. Následně uloží výsledek ve dvou částech do registrů EAX a EDX. Obdobně pro operaci dělení *div*. [TODO asm]

3.6.5 Instrukce pro bitovou rotaci a posuv

Tato skupina provádí operace bitového posuvu a rotace. Rotovat a posouvat lze vlevo nebo vpravo a také lze zvolit, zda je operace logická či aritmetická. [TODO asm]

Rozdíl mezi aritmetickým a logickým posuvem je ten, že zatímco u logické rotace vpravo je posloupnost bitů doplněna nulami zleva, u aritmetické rotace je posloupnost bitů doplněna tou hodnotou, kterou nabývá nejvíce významný bit. Kompilátory často v rámci optimalizace substituují operaci dělení bitovým posuvem. [TODO csprogrammer]

3.6.6 Instrukce pro předání kontroly

Instrukce *jmp* provede skok na specifikované návěští. Instrukce *call* slouží pro začátek vykonávání rutiny. Nejprve je obsah registru EIP zapsán na zásobník a poté se provede nepodmíněný skok na vybrané místo. Opačným způsobem postupuje instrukce *ret*, která slouží k návratu na místo, za kterým došlo k volání rutiny. [TODO asm]

```
SignedIsEQ_ proc
    push ebp
    mov ebp,esp

    xor eax,eax
    mov ecx,[ebp+8]
    cmp ecx,[ebp+12]
    sete al

    pop ebp
    ret
SignedIsEQ_ endp
```

Obrázek 10: Příklad programu ve strojovém jazyce. [TODO asm]

3.7 Překladače

Překlad z high-level jazyka do strojového kódu se odehrává v několika krocích, které bývají označovány jako *kompilační systém*. V první části se odehrává *preprocessing*, kde dochází k odstranění komentářů a ostatních znaků, které nejsou důležité pro běh programu, a také nahrazení direktiv pro vkládání knihoven kódem. V *kompilační* fázi dojde k převodu podoby kódu z high-level jazyka do jazyka symbolických adres. Zde může dojít k optimalizačním krokům, jako je například reorganizace kódu. Následující *assembly* část provádí překlad z jazyka symbolických adres do strojového kódu a zabalení do podoby *relocatable object programu*. Na závěr probíhá *linkování* dalších objektových souborů knihoven s naším souborem. [TODO csprogrammer]

Jelikož bude v rámci práce využit jazyk C/C++, následuje popis překladačů právě tohoto jazyka.

3.7.1 GCC

Zkratka GCC znamená *GNU Compiler Collection* a jedná se o integrovanou distribuci překladačů pro programovací jazyky jako je C, C++ či Fortran. [TODO gcc]

Překladač se při překladu programu drží dříve popsaneho postupu. Uživatel však může specifikovat množství nabízených možností. Běžná kompilace programu s jedním zdrojovým souborem by mohla vypadat následovně.

```
gcc -o hello.exe hello.c
```

Obrázek 11: Příklad běžné kompilace programu pomocí GCC. Zdroj vlastní.

Uživatel může proces kompilace zastavit po každé fázi. Pro zastavení po fázi *preprocessingu* slouží možnost *-E*. Výsledek je zobrazen na standardním výstupu, nebo ho můžeme zapsat do souboru pomocí možnosti *-o*. Pro zastavení po fázi *assembly* slouží možnost *-S*. Do souboru s příponou *.s* je uložena podoba našeho programu v jazyce symbolických adres. Pro zastavení po fázi *linkování* slouží možnost *-c*. [TODO gcc]

GCC překladač dovoluje specifikovat používaný standard jazyka C/C++ pomocí možnosti *-std*. Mezi podporované hodnoty patří *c90*, *c89*, *c11*, *gnu90*, *c++98*, či *iso9899:2017*. Dále je možné si vybrat, jaké uspořádání bitů struktur a unionů má být použito pomocí *-fstruct*, za který lze dosadit *big-endian*, *little-endian* nebo *native*. [TODO gcc]

Mezi důležité možnosti patří výběr agresivity optimalizace kódu. S využitím těchto možností se kompilátor snaží vylepšit výkon programu či snížit počet použitých instrukcí s daní delší doby kompilace. Optimalizace je možné zcela vypnout možností *-O0*. Mezi základní volby se řadí tři úrovně optimalizace, dané možnostmi *-O1*, *-O2* a *-O3*. Tyto příkazy zastřešují sadu dalších možností pro specifické případy. Mezi ně patří například možnost *-finline-functions*, která hodnotí vhodnost každé funkce programu pro *inlining*. *Inlining* nahrazuje kód pro volání funkce prostým vložením těla funkce do kódu. Další je možnost *-auto-inc-dec*, která kombinuje inkrementaci nebo dekrementaci adresy s přístupem do paměti. Zde je třeba myslet na to, že některé možnosti optimalizace jsou dostupné pouze na určitých architekturách. [TODO gcc]

3.7.2 Clang

„Překladač Clang je open-source kompilátor pro rodinu programovacích jazyků příbuzných s C a soustředí se na jejich nejlepší dostupnou implementaci. Staví na LLVM optimalizéru a generátoru kódu pro poskytnutí optimalizace vysoké kvality a generace kódu pro různé cíle.“ [TODO clang]

```
clang -o hello.exe hello.c
```

Obrázek 12: Příklad běžné kompilace programu pomocí Clang. Zdroj vlastní.

Clang nabízí rovněž extenzivní počet možností pro úpravu výsledku kompilace. Uživatel si opět volí několik optimalizačních úrovní, jak bylo popsáno dříve. Optimalizační schopnosti jsou dále rozšířeny o *optimalizace vedené profilováním*. Vylepšení spočívá ve sběru dat při prvotní kompilaci a vygenerování tabulky s těmito daty ve formátu podporovaném profilery. Profiler vygeneruje výstup, který popisuje, v jakém místě v kódu je stráveno nejvíce času a

tím pádem místo, kde mají optimalizace smysl. Rovněž mohou být zaznamenány informace o pravděpodobnostech větvení. [TODO clang]

3.7.3 MSVC

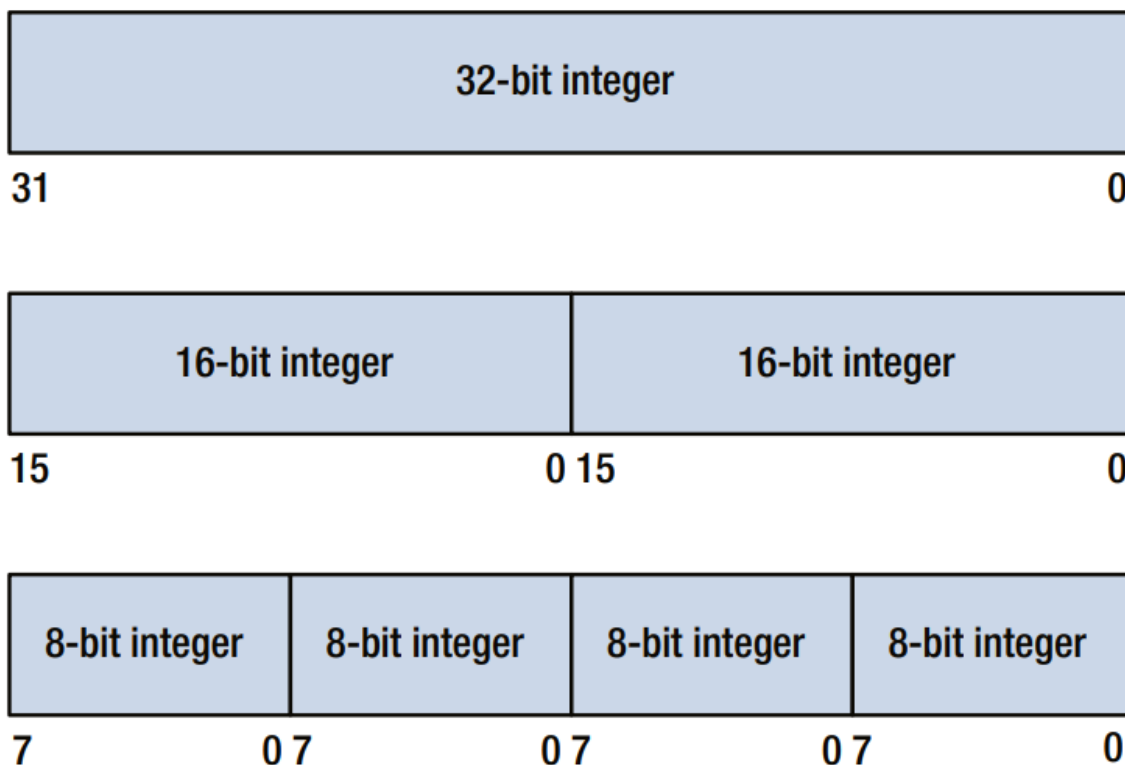
Microsoft Visual C++ je překladač vyvinutý společností Microsoft pro překlad programů psaných v C/C++ na platformě Windows. Překladač generuje soubory v Common Object File formátu a linker zase spustitelné soubory. [TODO msvcOptions]

```
cl hello.c
```

Obrázek 13: Příklad běžné kompilace programu pomocí MSVC. Zdroj vlastní.

3.8 SIMD

SIMD je způsob zpracování dat, kde je stejná operace aplikována na datovou sadu. Dnešní procesory dokáží s daty takto pracovat a programátorům nabízí celou řadu instrukcí pro aritmetické, logické operace a převody mezi typy. V roce 1997 byl do mikroarchitektury P5 od společnosti Intel zařazen výpočetní zdroj jménem MMX, který konceptu SIMD využívá a sloužil původně ke zrychlení operací nad multimediálními daty. [TODO asm]



Obrázek 14: Možnosti využití registru v SIMD. [TODO asm]

V případě že máme k dispozici 32-bitový registr, můžeme na něj nahlížet několika způsoby. Může obsahovat jedno 32-bitové číslo, dvě 16-bitová čísla nebo čtyři 8-bitová čísla. Díky tomu dokážeme provést tu samou operaci na více datech paralelně. Technologie MMX operuje s registry o velikosti 64 bitů. Později představená sada SSE rozšiřuje velikost na 128 bitů a v aktuální době nabízí největší velikost sada AVX, AVX2 a AVX-512, kde registry nabývají velikostí 256 a 512 bitů. [TODO asm]

255	128	127	0
	YMM 0		XMM 0
	YMM 1		XMM 1
	YMM 2		XMM 2
	YMM 3		XMM 3
	YMM 4		XMM 4
	YMM 5		XMM 5
	YMM 6		XMM 6
	YMM 7		XMM 7

Obrázek 15: Sada registrů AVX. [TODO asm]

Významnou roli u SIMD operací hraje zarovnání operandů v paměti. Data jsou zarovnána v případě, že jejich adresa je dělitelná velikostí jejich datového typu v bytech beze zbytku. Na rozdíl od fundamentálních datových typů, kde není zarovnání vyžadováno, při použití instrukčních sad SSE a AVX zarovnání dat vyžadováno je. I přesto se doporučuje zarovnávat fundamentální typy z důvodu možného zlepšení výkonu při přístupu do paměti. [TODO asm]

X86-SSE využívá pro výpočty 8 128-bitových registrů pojmenovaných *XMM0* až *XMM7*. Dále také obsahuje kontrolní a stavový registr *MXCSR*, který obsahuje příznaky po operacích s čísly s plovoucí řádovou tečkou. Máme zde k dispozici instrukce pro zpracování jak skalárních *single* a *double floating-point* čísel, tak takzvaných složených (*packed*) hodnot. Právě při využití složených hodnot lze využít paralelního zpracování více dat. Předtím je třeba je ovšem vložit do dříve zmíněných registrů. K tomu existují instrukce *movaps* či *movups*, které na rozdíl od MMX umožňují pracovat i s nezarovnanou pamětí. Poté můžeme použít instrukce pro běžné aritmetické a logické operace jako *addps*, *mulps*, *sqrtps* či *maxps*.

Navíc máme k dispozici další užitečné operace jako například horizontální součet sousedících hodnot ve zdrojovém a cílovém registru (*haddps*), změna pozic hodnot v rámci SSE registru pomocí bezprostřední hodnoty (*shufps*) nebo sloučení zdrojového a cílového registru pomocí bitové masky (*blendps*). [TODO asm]

V mnohých případech dokáže překladač sám vypořádat, kde by mohl aplikovat SIMD přístup ke zrychlení kódu. Příkladně sčítání dvou polí s celými čísly ve smyčce.

```
#include <stdint.h>
#include <stdlib.h>

#define N 8

int16_t* vectorAdd() {
    int16_t* a = aligned_alloc(16, sizeof(int16_t) * N);
    int16_t* b = aligned_alloc(16, sizeof(int16_t) * N);
    int16_t* res = aligned_alloc(16, sizeof(int16_t) * N);

    for (int i = 0; i < N; ++i) {
        res[i] = a[i] + b[i];
    }

    return res;
}

void main() {
    int16_t* res = vectorAdd();
}
```

Obrázek 16: Program sečtení dvou polí a uložení do výsledného pole. Zdroj vlastní.

Velikost datového typu (16), velikost pole (8) a hodnota zarovnání (16) byly zvoleny záměrně k demonstraci využití SSE. Tyto hodnoty zajistí, že se celé pole s celými čísly vejde do jednoho *XMM* registru a také, že bude zdrojové pole adekvátně zarovnáno. Nyní dokážeme pomocí adekvátních přepínačů u překladače GCC docílit využití operací nad složenými typy.

```
.L2:
    movzx    ecx, WORD PTR [rbx+rdx]
    add      cx, WORD PTR [rbp+0+rdx]
    mov      WORD PTR [rax+rdx], cx
    add      rdx, 2
    cmp      rdx, 16
    jne      .L2
```

Obrázek 17: Kód varianty s přepínači *-O1 -msse*. Zdroj vlastní.

```

movdqa    xmm0, XMMWORD PTR [rbp+0]
paddw     xmm0, XMMWORD PTR [rbx]
movaps     XMMWORD PTR [rax], xmm0

```

Obrázek 18: Kód varianty s přepínači *-O2 -msse*. Zdroj vlastní.

Jak můžeme vidět, úroveň optimalizace 1 ještě nestačí na exploataci SIMD dovedností. Teprve při využití druhé úrovně překladač zjistí, že tělo smyčky lze při využití složeného datového typu provést pomocí těchto 3 instrukcí.

```

.L2:
    movzx    ecx, WORD PTR [rbx+rdx]
    add      cx, WORD PTR [rbp+0+rdx]
    mov      WORD PTR [rax+rdx], cx
    add      rdx, 2
    cmp      rdx, 14
    jne      .L2

```

Obrázek 19: Kód varianty s přepínači *-O2 -msse* a lichým počtem prvků v poli.
Zdroj vlastní.

V případě, že počet prvků v poli je lichý, překladač nedokáže využít SIMD přístupu a výsledek je podobný jako v případě *-O1*.

Ve více komplikovaných příkladech je možné, že překladač nebude schopen převést kód na použití vektorových operací. V takovém případě je možné sáhnout po *wrapper* knihovnách, které poskytují abstrakce, které docílí vynucení SIMD operací, jako například Boost.SIMD. O krok níže k hardware je také možnost využití takzvaných intrinsických funkcí, které poskytují výrobci procesorů a existuje v nich řada funkcí jazyka C/C++, které se téměř jedna ku jedné mapují na SIMD instrukce procesoru. Jedna z takových knihoven je *immintrin.h*, která nabízí speciální datové typy a funkce s následující signaturou: `__m128i _mm_add_epi16 (__m128i a, __m128i b)`. [TODO - intrinsics] Jedná se o funkci, která využívá stejné instrukce jako v předchozím obrázku a slouží pro součet hodnot ve dvou *XMM* registrech, který obsahuje 8 16-bitových hodnot. Finální způsob spočívá v programování přímo pomocí jazyka symbolických adres a využívání požadovaných SIMD instrukcí.

3.9 Paralelizace

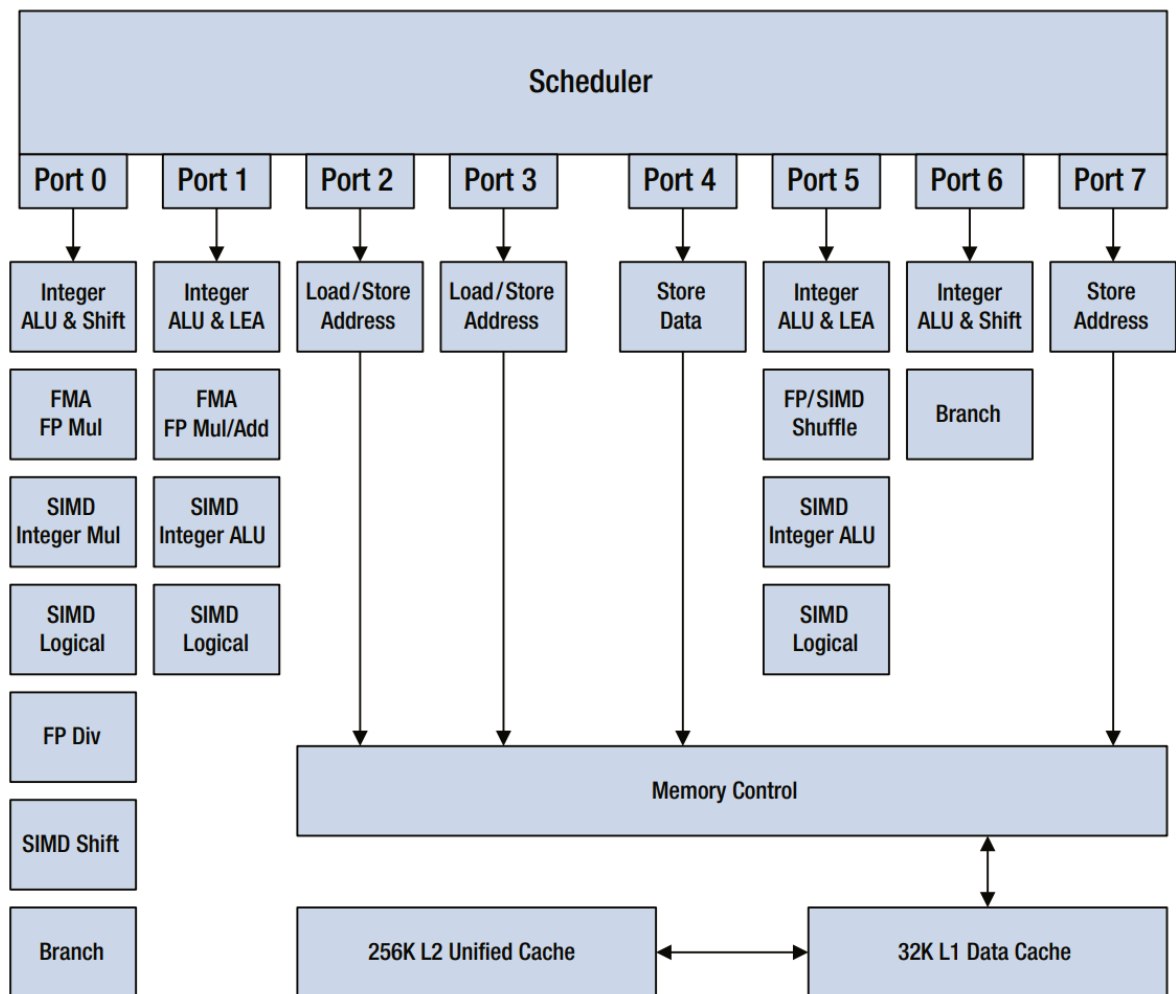
Paralelizace ve smyslu provádění výpočtů na více jádrech procesoru v jeden okamžik je v současné době hojně využívána a výrobci procesorů ji postupně rozvíjí z důvodu přiblížení

k fyzikálním hranicím integrace na čipu. Oproti vývoji software v minulosti může poskytnout značný posun ve výkonech aplikací, ale také s sebou nesou svá specifika a potíže. Je však třeba si uvědomit, že v rámci výpočetního systému se paralelizace vyskytuje v různých podobách na různých úrovních abstrakce.

3.9.1 Instruction-level paralelismus

Využití paralelismu na úrovni instrukcí je umožněno existencí a využití pipeline při vykonávání instrukcí. K potenciálnímu zrychlení dochází kvůli faktu, že instrukce mohou být prováděny paralelně. Snaha využít paralelismus na této úrovni může být zprostředkován jak hardwarově, v podobě dynamického hledání využití paralelismu při běhu programu, tak softwareově, kde je potenciál paralelizace stanoven staticky při psaní kódu. Ukazatelem efektivity pipeline může být *CPI*, který udává počet cyklů potřebných pro provedení instrukce a pro výpočet využívá také hodnoty potencionálních pozastavení z důvodu *data* či *control hazards*. Mnoho optimalizací je zprostředkováno překladačem, který se snaží efektivně využít pipeline. [TODO caqa]

Pro pochopení této úrovně paralelismu je třeba si uvědomit, že skutečný způsob vykonávání instrukcí je dán mikroarchitekturou daného procesoru, který podporuje i více než 100 operací *in-flight* v jednom časovém okamžiku díky využití technologií jako CPU pipelining či *out-of-order execution*. [TODO csprogrammer]



Obrázek 20: Exekuční engine Haswell CPU. [TODO asm]

Z důvodu vícenásobného počtu exekučních jednotek pro tu samou operaci můžeme docílit souběžného vykonávání více mikro-operací stejného typu.

3.9.1.1 Loop unrolling

Jedná se o přístup replikace těla smyčky s cílem zvýšit počet užitečných instrukcí, které skutečně provádí nějaký pro nás významný výpočet, a také lépe uspořádat po sobě následující instrukce do pipeline s využitím navzájem nezávislých operací. [TODO caqa]

Po překladu programu, ve kterém se nachází iterace ve smyčce, do jazyka symbolických adres běžně vidíme využití instrukcí podmíněného skoku a dekrementace počítadla pro zprostředkování očekávaného chování. Tyto dvě instrukce však mohou mít za následek pozastavení pipeline. Několikanásobnou replikací těla smyčky vzniká potenciál vstupu operací z po sobě jdoucích iterací do pipeline, pokud jsou na sobě nezávislé. Tato optimalizace navíc

otevřít dveře pro aplikaci dalších optimalizací, jako například přeskládání výrazů. [TODO caqa]

Při používání tohoto způsobu pro zrychlení běhu programu je třeba uvažovat nad tím, kolikrát by mělo být tělo smyčky replikováno. Od určitého počtu už by mohlo docházet k poklesu výkonu. Rovněž dojde k nárůstu velikosti souboru a následně k potencionálním *cache miss* v instrukční vyrovnávací paměti. Dalším limitujícím faktorem je počet registrů, který musí být využit. Z důvodu zachování nezávislosti instrukcí musí být mezivýsledky ukládány do odlišných registrů, jejichž počet je omezený. [TODO caqa]

3.9.1.2 Předpovídání větvení

Větvení v programu může mít za následek snížení výkonu z důvodu špatného odhadu cesty. V předchozím případě jsme toto riziko snížili díky snížení počtu podmíněných skoků. Z důvodu potřeby lepšího odhadování při skocích jsou využívány různé druhy předpovídání. Užitečné informace mohou být zjištěny ze statické analýzy kódu, pro razantní snížení penalizace za špatnou předpověď je však zapotřebí využít také dynamické předvídání. Zkoumání výsledků nedávných větvení je aplikováno *korelačním prediktorem*, který podle posbíraných výsledků rozhodne cestu aktuální větve. Je možné také sledovat celou historii větvení pomocí zápisu do posuvného registru, jehož každý bit určuje, kterým směrem jsme se vydali v odpovídající větvi programu. [TODO caqa]

Zlepšit výkon při větvení můžeme několika způsoby. První z nich lze aplikovat na podmínky a spočívá v provádění porovnání, jehož výsledek lze snadno odhadnout. Alternativně lze upravit způsob programování a využít přístup podmíněných instrukcí, jako je například *cmovge*. Těto instrukce předchází porovnání hodnot dvou registrů a přesun dat je proveden pouze v případě, že hodnota zdrojového registru je větší nebo rovna hodnotě cílového registru. Díky tomuto přístupu můžeme dosáhnout konstantního výkonu nezávisle na složitosti odhadu výsledku porovnání. [TODO csprogrammer]

3.9.2 Data-level paralelismus

Mluvíme-li o paralelismu na úrovni dat, máme na mysli především SIMD výpočetní model. Můžeme sem také zařadit výpočty na procesorech s vektorovou architekturou, či GPU. Podnětem pro rozšíření SIMD exekučních jednotek a instrukcí byl nárůst počtu aplikací, které by právě takové zpracování dat mohly využít. Mezi ně lze zařadit práci s multimediálními daty a maticemi. SIMD má potenciál být energeticky úsporné, jelikož jedna instrukce

zpracuje více dat najednou. Navíc náročnost hardwarové implementace takovýchto operací není velká. [TODO caqa]

3.9.2.1 *Strip mining*

Jak je zřejmé z dříve popisovaného příkladu převodu kódu s použitím SIMD instrukcí, nelze vždy spoléhat na překladač. Je třeba myslet na to, že tyto vektorové instrukce pracují s registry o určité délce, která je zpravidla mocnina dvou, a také na zarovnání v paměti. Zarovnání je důležité i z toho důvodu, aby se vektor v hodnot paměti nenacházel na dvou různých paměťových stránkách. Délka pole, se kterým programátor pracuje, zřídka odpovídá požadavkům. Navíc ne vždy ji dokážeme dopředu odhadnout. V takových případech však může programátor vynaložit úsilí, aby překladači práci usnadnil. [TODO caqa]

Pokud známe maximální povolenou délku registru pro vektorové operace, dokážeme si data a smyčky nachystat tak, aby bylo data možné na registry namapovat. Takový postup je označován jako *strip mining*. Pokud se v původním programu vyskytovala jedna smyčka, rozdělíme ji na dvě. Hlavní smyčka zpracuje data, kterými lze plně naplnit vektorové registry a vedlejší smyčka, která zpracuje zbytek. [TODO caqa]

3.9.2.2 *Výpočty na GPU*

Společnost Nvidia, zabývající se převážně vývojem grafických čipů, představila programovací jazyk CUDA pro využití paralelního potenciálu svých karet. Programátor zde využívá SIMD principů na masivní úrovni paralelizace, ale musí se také seznámit se specifikami GPU platformy. Programátorům jsou zde k dispozici abstrakce s názvy *thread*, *block* a *grid*. Tyto pojmy se dají přirovnat ke běžně používaným termínům ze všedních programovacích jazyků. *Thread* si můžeme představit jako jednu právě probíhající iteraci smyčky. Ty se dále seskupují do *bloku*, který můžeme přirovnat tělu smyčky. *Bloky* jsou dále uspořádávány do *gridu*, který představuje vektorizovatelnou smyčku. Počty vláken v bloku a bloků v mřížce si při výpočtu stanovuje programátor. Takovéto bloky jsou posílány plánovačem na vícevláknový SIMD procesor pro vykonání. [TODO caqa]

3.9.2.3 *Loop-level parallelismus*

Konstrukt označovaný jako smyčka je v mnoha ohledech vhodným kandidátem na paralelizaci na více různých úrovních souběžnosti. To ovšem platí pouze při dodržení určitých pravidel, které do velké míry diktuje hardware. Značné množství typů závislostí může být odhaleno statickou analýzou kódu pro smyčku, mezi které patří například *loop-carried*

dependence. Tato závislosti popisuje využití dat vypočítaných v předchozích iteracích iteracemi následujícími. [TODO caqa]

```
for (i=0; i<100; i=i+1) {  
    A[i] = A[i] + B[i];    /* S1 */  
    B[i+1] = C[i] + D[i]; /* S2 */  
}
```

Obrázek 21: Ukázka *loop-carried* závislosti. [TODO caqa]

Příklad ukazuje závislost na předchozí iteraci u pole B. První výpočet závisí na hodnotě, která byla vypočtena v předchozí iteraci. Jelikož neexistuje cirkulární závislost mezi těmito dvěma výpočty, smyčka může být paralelizována. [TODO caqa]

```
A[0] = A[0] + B[0];  
for (i=0; i<99; i=i+1) {  
    B[i+1] = C[i] + D[i];  
    A[i+1] = A[i+1] + B[i+1];  
}  
B[100] = C[99] + D[99];
```

Obrázek 22: Přepis smyčky. [TODO caqa]

Jelikož oba výpočty v rámci jedné iterace nejsou závislé na sobě, můžeme změnit jejich pořadí. V nulté iteraci se počítá s hodnotou pole B na pozici nula, Tento výpočet můžeme přesunout před smyčku. [TODO caqa]

Překladače kontrolují existující smyčky v programu a nalézají závislosti pomocí stanovení, zda jsou indexy pro přístup k prvkům pole *afinní*. *Afinní* index lze vyjádřit ve formátu $a * i + b$, kde a a b jsou konstanty, a i je iterační proměnná. Jestliže mají dvě afinní funkce stejnou hodnotu při různých hodnotách indexu, existuje závislost. Pokud je jiný způsob přístupu do toho samého pole hodnot vyjádřen jako $c * i + d$, můžeme závislost odhalit také pomocí největšího společného dělitele. V případě, že $GCD(c,a)$ dělí $(d - b)$, existuje *loop-carried* závislost. [TODO caqa]

3.9.3 Thread-level paralelismus

Postupem vývoje výpočetní techniky začali vývojáři narážet na fyzikální hranice tehdejších procesorů a také klesající výtěžnost z exploatace ILP. Kvůli nikdy nekončícímu hledání vyššího výpočetního výkonu byly vyvinuty jak procesory obsahující více než jedno výpočetní jádro (*multicore*), tak systémy obsahující více procesorů (*multiprocessor*). Multiprocesorové systémy jsou charakterizovány úzce svázanými procesory, jejichž koordinace je zajištěna operačním systémem a využívají sdíleného paměťového prostoru. U tohoto přístupu je využíván princip paralelního zpracování ve smyslu spolupráce více vláken na jednom úkolu, nebo existence navzájem nezávislých procesů, což je označováno jako *request-level paralelismus*. [TODO caqa]

3.9.3.1 Amdahlův zákon

Amdahlův zákon popisuje možné zrychlení běhu algoritmu při představení vylepšení v libovolné podobě. Dosažitelné zrychlení je omezeno podílem času, ve kterém může být vylepšení využito. [TODO caqa]

Zrychlení je vyjádřeno jako:

$$\text{Zrychlení} = \frac{\text{Doba běhu algoritmu bez vylepšení}}{\text{Doba běhu algoritmu s vylepšením tam, kde může být využito}} \quad (1)$$

Celkové zrychlení závisí na dvou faktorech. Prvním z nich je podíl času běhu částí s a bez vylepšení, označován jako Zrychlení_V . Dalším z nich je určení, jak velká část programu může toto vylepšení využít, což označíme jako Podíl_V . Nová doba běhu algoritmu je tedy dána jako:

$$\text{Doba}_N = \text{Doba}_S * \left((1 - \text{Podíl}_V) + \frac{\text{Podíl}_V}{\text{Zrychlení}_V} \right), \text{ kde} \quad (2)$$

- Doba_N je doba běhu algoritmu s vylepšením
- Doba_S je doba běhu algoritmu bez vylepšení

Celkové zrychlení je vyjádřeno jako:

$$\text{Zrychlení}_C = \frac{\text{Doba}_S}{\text{Doba}_N} = \frac{1}{(1 - \text{Podíl}_V) + \frac{\text{Podíl}_V}{\text{Zrychlení}_V}} \quad (3)$$

Tento zákon vyjadřuje klesající výtěžnost implementace nových vylepšení. Velikost části programu, na kterou může být aplikováno vylepšení, diktuje maximální možné dosažitelné zrychlení. [TODO caqa]

3.9.3.2 *Koherence vyrovnávacích pamětí*

U *multicore* systému můžeme pozorovat asymetrický přístup do paměti. Načítání dat z vyrovnávací paměti je rychlejší než načítání dat paměti operační. Při využití více jader procesoru k provádění výpočtů můžeme data rozdělit na *privátní* a *sdílená*. *Sdílená* data jsou jednou z forem komunikace mezi jádry. Ukládání takovýchto dat do vyrovnávací paměti jader představuje problém koherence vyrovnávacích pamětí. Koherence vyrovnávacích pamětí zajišťuje, že více procesorů má konzistentní pohled na paměť. Paměťový systém označujeme za koherentní v případě, že čtením libovolných dat získáme nejaktuálnější data. Pro zajištění této vlastnosti jsou aplikovány dva přístupy. První je označován jako *snooping*, kde každá vyrovnávací paměť, která obsahuje určitý blok paměti sleduje, zda je tento blok sdílen, pomocí čtení sdíleného vysílacího média. Další se jmenuje *directory based*, který soustřeďuje informace o tom, zda jsou bloky paměti sdíleny, do jednoho místa. Při využití *snooping* protokolu, první procesor, který zapíše do paměťového bloku, se stane jeho *vlastníkem*. Tento způsob bývá označován jako *write-invalidate*. Pokud jádra chtějí zapisovat do sdíleného bloku, nebo se jádro pokusí o čtení modifikovaných dat v bloku, může dojít k takzvanému *true sharing miss*. Podobné chování nastává také při *false sharing miss*, kdy zápis jednoho jádra do *cache bloku* způsobí invalidaci dat jádra jiného, aniž by se odkazovala na ta samá data. [TODO caqa]

3.9.3.3 *Synchronizační primitiva*

Jedním z přístupů pro vynucení sekvenčních zápisů je využití synchronizačních primitiv poskytovaných hardwarem. Toto řešení může mít za následek zvýšení latence operací s vysokou úrovní sdílených dat. Pokud to ISA nabízí, lze využít párových instrukcí *load linked* a *store conditional*. Jestliže je po vykonání načtení změněna hodnota v paměti, odkud načtení proběhlo, pak instrukce pro zápis selže. To samé se stane i v případě změny kontextu procesoru. [TODO caqa]

```
try:    MOV R3,R4    ;mov exchange value
        LL  R2,0(R1);load linked
        SC  R3,0(R1);store conditional
        BEQZ R3,try ;branch store fails
        MOV R4,R2    ;put load value in R4
```

Obrázek 23: Příklad atomické výměny hodnot. [TODO caqa]

Běžně používaným primitivem je zámek. Jelikož je při žádání o zámek možné, že je aktuálně zabraný jiným jádrem, musí se jádro, které zámek žádá, točit ve smyčce a zkoušet žádat opakovaně. Proto se tomuto přístupu také říká *spin lock*. Při využití koherenčního protokolu využíváme dvou principů. Prvním z nich je, že kontrola stavu zámku může být prováděna na lokální kopii dat v cache jádra. Druhým je, že jestliže jádro využilo zámku v současnosti, je pravděpodobné, že tak učiní znovu. [TODO caqa]

```
lockit:  LDR2,0(R1)      ;load of lock
        BNEZR2,lockit    ;not available-spin
        DADDUIR2,R0,#1    ;load locked value
        EXCHR2,0(R1)      ;swap
        BNEZR2,lockit    ;branch if lock wasn't 0
```

Obrázek 24: Spin lock. [TODO caqa]

V případě současné existence tří jader J0, J1 a J2 bude postup vypadat následovně. J0 vlastní zámek a J1 a J2 ve smyčce kontrolují své lokální kopie sdíleného zámku. Jakmile J0 zapíše hodnotu 1, zámek se uvolní a J0 se stane majitelem zámku, což invaliduje lokální kopie v cache zbylých jader. Obě jádra vygenerují *cache miss* a následně požádají o opětovné načtení zámku do cache, který bude nyní opět sdílený. Jedno z jader je obslouženo jako první, a právě to bude moci zámek zamknout. Toto jádro zapíše 1 do zámku pomocí atomické výměny a jestliže je načtená hodnota rovna 0, zámek může být zamčen aktuálním jádrem, a to může pokračovat do kritické sekce. Druhé jádro opět zapíše 1 do zámku, ale protože mu atomická výměna vrátí hodnotu 1, vrací se na začátek smyčky. [TODO caqa]

3.9.3.4 POSIX vlákna

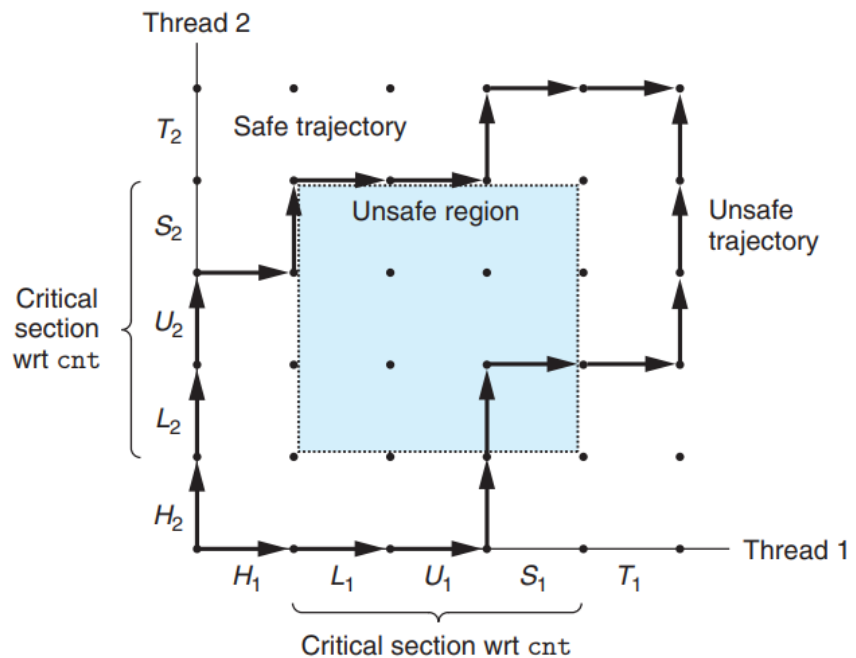
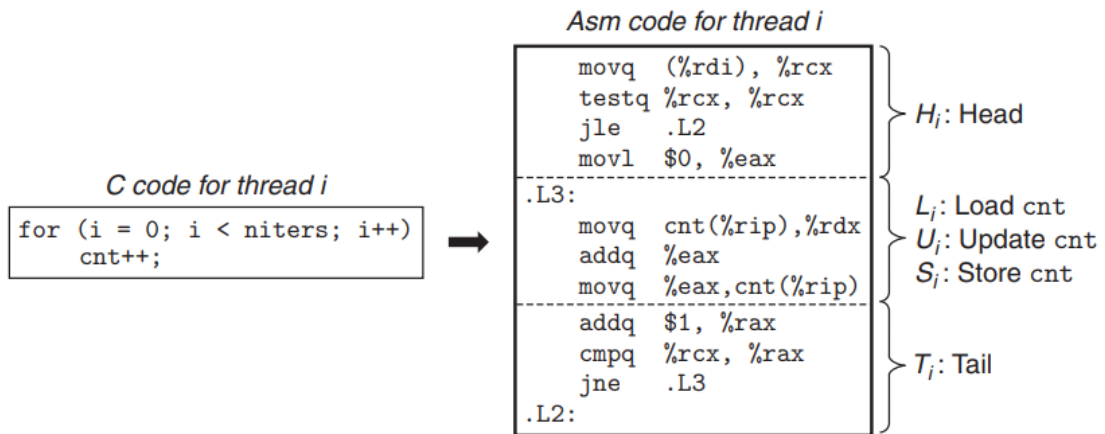
Proces je abstrakce operačního systému pro právě vykonávající se program. Pracuje s iluzí, že mu náleží všechny prostředky procesoru a také paměti. Zároveň se jeví, že jeho instrukce

jsou vykonávány jedna po druhé bez přerušení. Právě vykonávající se program má svůj *kontext*, který je mimo jiné určen daty a kódem programu v paměti a hodnotami v CPU registrech. *Vlákno* je logický proud instrukcí vykonávající se v kontextu procesu. Každé vlákno má svůj kontext, identifikátor, zásobník, ukazatel na vrchol zásobníku, stavový registr a registry k obecnému použití. [TODO csprogrammer]

Mezi vlákny nemusí existovat rodičovská hierarchie. Můžeme je rozdělit na *hlavní* a *vrstevnická*. Každý proces začíná s jedním hlavním vláknem a po čas existence může vytvářet vrstevnická vlákna. Po dobu jejich existence dochází ke střídání vykonávání těchto vláken. [TODO csprogrammer]

POSIX vlákna poskytují standardní rozhraní pro práci s vlákny v jazyce C. Funkce pro vytvoření vlákna `pthread_create` slouží k vytvoření jádra. Můžeme jí jako argumenty poslat ukazatel na funkci, kterou chceme ve vlákně vykonávat, spolu s argumenty pro tuto funkci. Na zpět obdržíme identifikátor vlákna. Všechna existující vlákna mohou být zrušena voláním funkce `pthread_exit` a pro jednotlivá vlákna slouží `pthread_cancel`. Jestliže od vlákna očekáváme návratovou hodnotu, po dokončení jeho běhu obdržíme výsledek pomocí `pthread_join`, které předáme identifikátor vlákna a ukazatel pro data. [TODO csprogrammer]

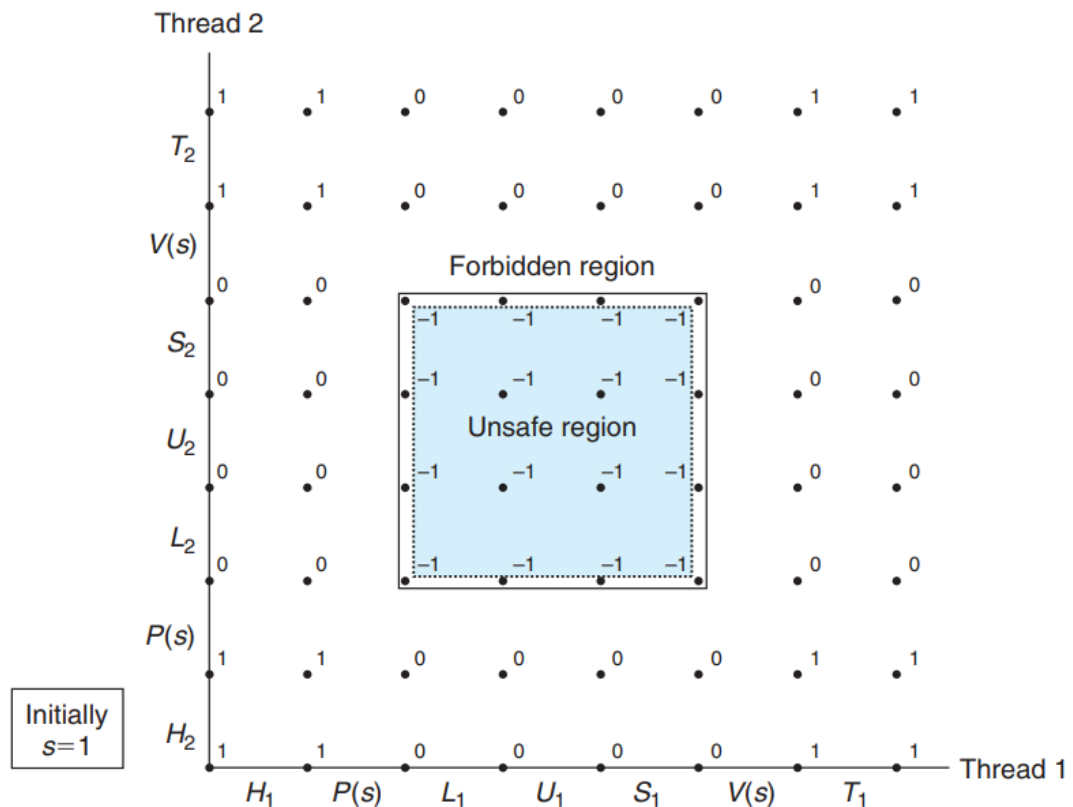
Opět je třeba myslet na potíže nastávající při využití sdílených proměnných ve vláknech. Mezi ně se řadí globální proměnné, statické proměnné a také lokální proměnné předané jako argument pomocí ukazatele. Po dobu současné existence více vláken nedokážeme předpovědět, kdy dojde ke změně kontextu. Pokud více vláken zapisuje do sdílené proměnné, její hodnoty s velkou pravděpodobností nebudou odpovídat očekávání a také budou v různých bězích programu nekonzistentní. V posloupnosti instrukcí programu se nachází skupiny navazujících instrukcí, při jejichž vykonávání by nemělo dojít ke změně kontextu, protože tyto instrukce nějakým způsobem pracují se sdílenou proměnnou. Těmito skupinám se říká *kritická sekce* a můžeme si ji zobrazit v *grafu průběhu*. [TODO csprogrammer]



Obrázek 25: Graf průběhu. [TODO csprogrammer]

3.9.3.5 Semafor

Jedním z nástrojů synchronizace vláken je *semafor*. Jedná se o globální proměnnou, která obsahuje číselnou hodnotu. Tato hodnota je manipulována funkcemi $P(s)$ a $V(s)$. Jestliže voláme funkci P a proměnná s je větší než 0, žádající vlákno ji sníží o 1 a pokračuje. Jinak čeká, až proměnná s nabyde hodnoty 0. Při volání funkce V se hodnota s zvyšuje o 1, čímž dochází k odblokování právě jednoho vlákna. V případě, že je iniciální hodnota s rovna 1, mluvíme o *binárním semaforu*, či *mutexu*. Využití takového synchronizačního objektu můžeme promítnout do grafu průběhu. [TODO csprogrammer]



Obrázek 26: Graf průběhu s promítnutými hodnotami binárního semaforu.

[TODO csprogrammer]

Tuto synchronizační strukturu můžeme také využít pro signalizaci dostupnosti sdílených prostředků. Toho docílíme inicializací semaforu na hodnotu větší než 1. Toto lze ilustrovat na příkladu *producent-konzument*, kde mimo výlučný přístup ke sdílenému datovému bufferu musíme také zajistit uspořádaný přístup. Proto zde ještě musíme použít další dva semafore. Jeden semafor pro počítání volných míst v bufferu a jeden pro počítání položek umístěných v bufferu. Pokud chce producent přidat položku do bufferu, musí jako první zkontrolovat semafor signalizující počet volných míst. Jakmile se uvolní místo, musí zkontrolovat semafor pro přístup ke sdílenému bufferu. Poté může položku přidat a signalizovat odemčení semaforu bufferu a také navýšení počtu položek v bufferu pomocí druhého semaforu. Konzument na rozdíl od toho jako první kontroluje semafor pro počet položek v bufferu a poté inkrementuje semafor pro volná místa. [TODO csprogrammer]

4 MĚŘENÍ VÝKONU APLIKACÍ

Budeme-li hovořit o výkonu počítačového programu, různí lidé si pod tímto pojmem mohou představit odlišné metriky. Významná metrika označovaná jako *doba vykonávání* (*execution time*) udává čas mezi začátkem a koncem události. Doba vykonávání programu je spolehlivým ukazatelem výkonu a používání jiných metrik může vést k zavádějícím závěrům při optimalizacích. [TODO caqa]

Na úrovni procesoru sledujeme časové jednotky pojmenované *cykly*, které se využívají k řízení a synchronizování operací. Délka cyklu může být odvozena z taktu procesoru, který v současnosti nabývá hodnot větších než 3 GHz na běžně dostupných procesorech. Známe-li počet cyklů pro provedení celého programu, zjistíme *procesorový čas*. [TODO caqa]

$$\text{Procesorový čas} = \frac{\text{Počet cyklů programu}}{\text{Takt procesoru}} \quad (4)$$

V případě, že známe počet instrukcí v programu, zjistíme průměrný počet cyklů na instrukci, označováno jako CPI. [TODO caqa]

$$\text{CPI} = \frac{\text{Počet cyklů programu}}{\text{Počet instrukcí programu}} \quad (5)$$

4.1 Benchmarking

Návrháři výpočetního hardware využívají zátěžové testy, zvané *benchmarks*, pro posouzení vhodnosti navrženého hardware pro určité operace. Benchmark má podobu krátkého programu, který obsahuje malou část programu z reálné aplikace, či většího programu imitujícího chování reálné aplikace. Společnosti, jako například SPEC, se soustředí na vývoj benchmarkových sad, které by měly co nejlépe a nejobjektivněji posoudit výkon hardware. [TODO caqa]

4.1.1 Microbenchmark

Microbenchmark měří výkon malé části programu. Nemohou být použity pro posouzení celkového výkonu aplikace, ale jsou vhodné pro posouzení latence operací a propustnosti. [TODO micro]

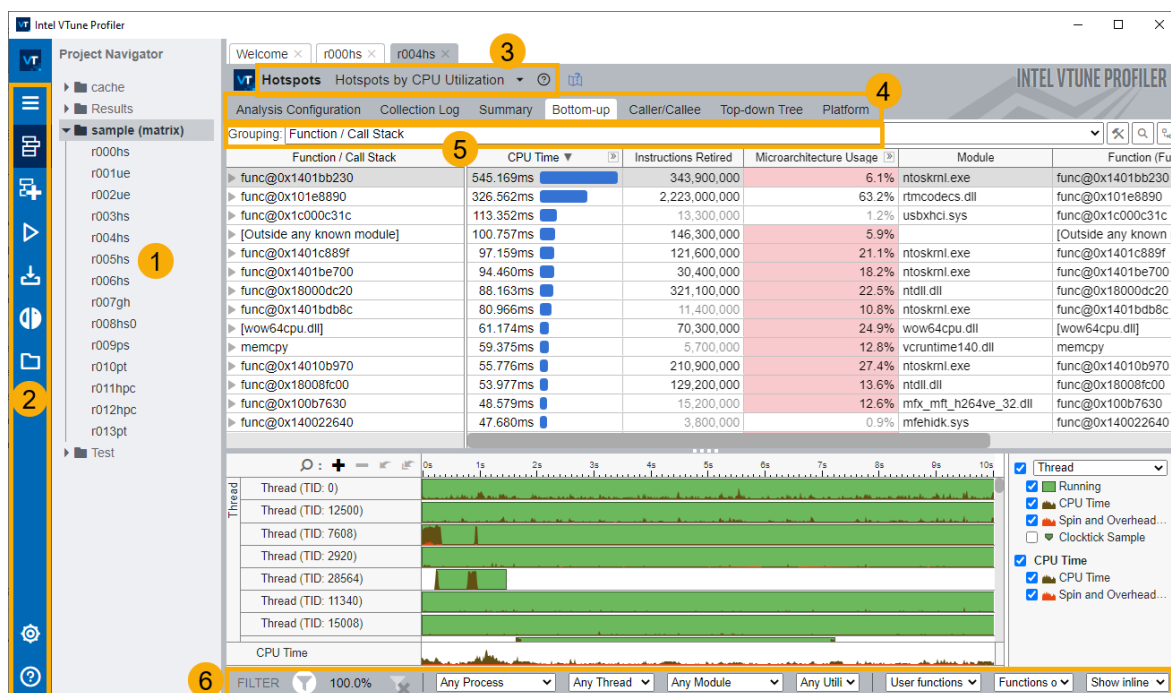
V rámci této práce bude využita knihovna Google Benchmark, která slouží k ohodnocení výkonu funkcí v programu, podobně jako u jednotkových testů.

4.2 Profilování

Optimalizace se mimo jiné zabývá snižováním doby běhu programu. Abychom zjistili, proč program neběží tak rychle, jak by mohl, používáme profilování. K tomu slouží nástroje zvané *profilery*, které nám umožňují měřit určité metriky a díky nim dokážeme pochopit výkonnostní charakteristiky programu. Tyto nástroje využijeme jak pro měření výkonu, tak při hledání kritických míst v programu při optimalizaci. Významnou částí těchto nástrojů je možnost vizualizace průběhu programu. [TODO profile]

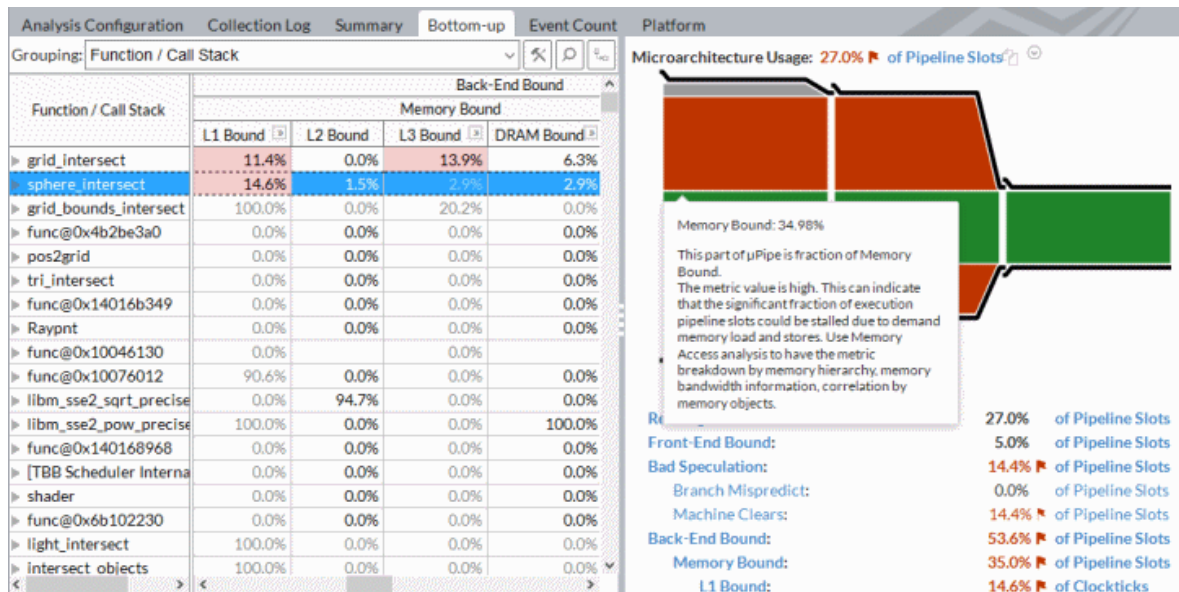
4.2.1 Vzorkovací profilery

Vzorkovací profilery fungují na principu periodického čtení informací ze zásobníku volání. Frekvence přerušování může být dána uživatelem. Tyto nástroje pracují s předpokladem, že pokud je doba běhu funkce delší než doba běhu ostatních funkcí, při načtení dat profilerem se budeme v této funkci nacházet častěji než v ostatních funkcích. [TODO profile]



Obrázek 27: Grafické prostředí programu Intel VTune. [TODO vtune-guide]

Profilovací nástroj Intel VTune nám nabízí množství funkcí pro analýzu různých metrik. V adresářové struktuře u čísla 1 se nachází záznamy jednotlivých profilovacích relací. Aplikace nabízí různé druhy profilovacích běhů, které mohou být dopodrobna zkoumány přepínáním mezi záložkami označenými u čísla 4. V comboboxu níže můžeme zvolit předem nastavené skupiny dat, které chceme sledovat, a také granularitu těchto skupin pro zjištění různých pro nás významných informací. [TODO vtune-guide]

Obrázek 28: Okno s μ pipe zobrazením. [TODO vtune-guide]

Jednou z metod analýz je prozkoumání využití mikroarchitektury procesoru. Aplikace využívá metrik získaných na základě událostí. Hodnocení je dáno na základě sledování využití pipeline procesoru. Tuto pipeline můžeme rozdělit na *front-end*, ve kterém se odehrává načítání a dekodování instrukcí, a *back-end*, který popisuje samotné vykonávání instrukcí za použití funkčních jednotek procesoru. Za předpokladu, že víme, že front-end v každém cyklu může vygenerovat až 4 operace, můžeme porovnat naměřená data s daty ideálními. Jestliže volná místa v pipeline nebyla zcela naplněna, protože například nedošlo ke včasnému načtení instrukcí, je tato skutečnost označena jako *front-end bound*. Pokud nemohl back-end přijmout více operací stejného typu, mluvíme o *back-end bound*. Ve výše zobrazeném barevném grafu nám aplikace ukazuje jak dobře nebo špatně je pipeline využita a v jaké části je problém. Je zobrazeno, jaké procento instrukcí bylo načteno a provedeno (*retired*), načteno a zahozeno (*bad speculation*) a také front-end a back-end bound. Tyto kategorie jsou popsány *top-down* modelem a každá z nich má stanovenou své limitní procento, které je při překročení označeno a může pomoci identifikovat potencionální problém. Alternativní pohled *bottom-up* poskytuje výčet identifikovaných objektů ze zásobníku volání, jako třeba programové funkce nebo moduly, a uvádí dostupné naměřené metriky. Opět jsou zvýrazněny překročené hranice předem nastavených limitů. Odsud dokážeme nalézt i konkrétní řádek kódu, který má na svědomí zpomalení. [TODO vtune-guide]

Analysis ConfigurationCollection LogSummaryBottom-upEvent CountPlatformsphere.cpp

SourceAssembly

S... ▲	Source	🔥 Clockticks	Instructions Retired	CPI Rate	Locators			
					Retiring	Front-End	Bad Speculation	Back-End Bound
112	VDOT(b, V, zy->d);	580,800,000	900,000,000	0.645	4.5%	0.5%	5.0%	0.1%
113	VDOT(temp, V, V);							
114								
115	disc=b*b + spr->rad*spr->rad - temp;	1,550,400,000	1,936,800,000	0.800	10.8%	1.7%	5.9%	26.7%
116								

Obrázek 29: Pohled *source code analysis*. [TODO vtune-guide]

Aplikace pro získávání profilovacích dat využívá jednotek pro monitorování výkonu (*PMU*), což jsou hardwarové jednotky nacházející se na čipu a mají za úkol sledovat počty určitých událostí, jako je třeba počet cache-miss. Tyto jednotky spouštějí řadu událostí, které lze odposlouchávat. Aplikace označuje funkce, které zabírají nejvíce procesorového času, jako *hotspots*, na jejichž optimalizaci bychom se měli zaměřit. Pro detailnější identifikaci problémů můžeme využít dalších typů analýzy. Můžeme dále sledovat *memory bound* potíže, způsobené například cache-misses, nebo *core bound*, což popisuje suboptimální využití exekučních jednotek procesoru. [TODO vtune-cook]

4.2.2 Instrumentační profilery

Instrumentační profilery využívají speciálních částí kódu, často v podobě maker, pro vlastní označení sekcí našeho kódu. Na rozdíl od vzorkovacích profilerů jsou tyto nástroje přesnější, protože nepracují s průměrnými výsledky. Jelikož je tato metoda v rukou uživatele, může profileru mimo jiné poskytnout další data, která považuje za významná ke sledování. [TODO profile]

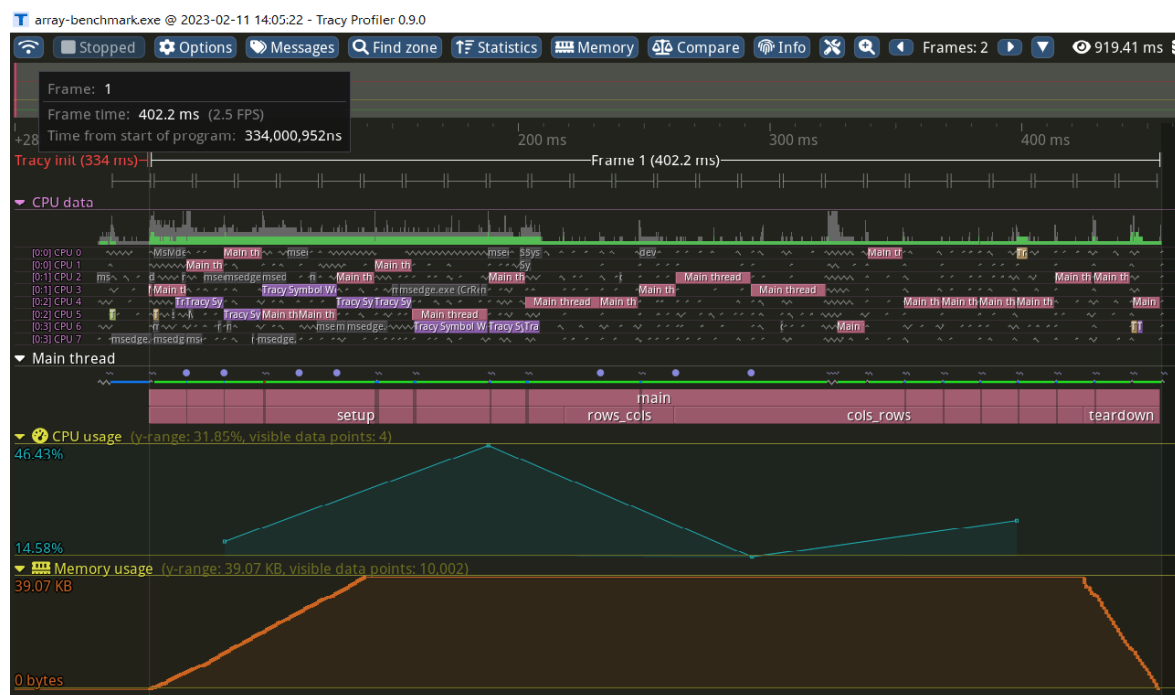
Zástupcem této kategorie profilovacích nástrojů je program Tracy. Jedná se o hybridní snímkový a vzorkovací profiler s nanosekundovým rozlišením. Na rozdíl od ostatních instrumentačních nástrojů, dokáže také poskytnout statistické informace na základě dat ze zásobníku volání, podobně jako vzorkovací profilery. Tento nástroj jen velmi málo zatěžuje profilovaný program voláním svých funkcí, protože zaznamenání profilovací události trvá jenom pár nanosekund. Tracy považuje *snímek (frame)* jako základní pracovní jednotku při profilování. Pojem snímek je použit proto, že se od programu očekává, že bude použit pro profilování výkonu her. Snímky však mohou být aplikovány univerzálně a jejich použití navíc není vynuceno. Program dovoluje uživateli používat jak rozhraní v příkazové řádce, tak grafické rozhraní. V obou způsobech použití komunikuje profilovaná aplikace s profilovacím nástrojem prostřednictvím síťové komunikace. Aplikaci lze sledovat v reálném čase, také po

jejím skončení, nebo může uživatel do aplikace načíst dříve zaznamenané profilovací logy. [TODO tracy]

Knihovna Tracy poskytuje programátorovi množství způsobů, jak zaznamenat důležitá aplikační data. Funkce *TracyMessageL* bere jako argument textový řetězec, který následně zobrazí v profileru v odpovídajícím čase. Zprávy je možné barevně odlišit. Snímek je označen pomocí *FrameMark*. K volání by mělo dojít ihned po vyrenderování snímku. Mimo funkce užitečné pro profilování her, existuje také příkaz *ZoneScoped*, který sleduje danou zónu. Tento příkaz funguje pomocí sledování speciální profilovací proměnné umístěné na zásobníku. Běžné použití je na začátku vykonávání funkcí, které chceme sledovat, ale příkaz může být umístěn také do smyček. Pomocí *TracyPlot* můžeme profilovací aplikaci posílat data a sledovat jejich postupný vývoj v čase. Rovněž dokážeme sledovat využití paměti pomocí funkce *TracyAlloc* a *TracyFree*. Těmto funkcím je poskytována proměnná typu ukazatel. Při používání tohoto mechanismu je třeba myslet na to, kdy ukazatel předat. Profilovací funkci zaznamenávající alokaci musíme zavolat po alokaci paměti a uvolňovací funkci zase před samotným uvolněním paměti. Dále existují funkce pro profilování aplikace využívající technologii OpenGL, Vulkan, Direct3D 11, Direct3D 12 a OpenCL. Výše uvedené příkazy mohou být obohaceny o data ze zásobníku volání v případě přidání písmene *S* za název funkce. Pak je třeba specifikovat hloubku, která se má sledovat. Jestliže je profilovací aplikace spuštěna pod zvýšenými právy, dochází k automatickému sledování rozšířených informací. Sem patří například využití a topologie CPU, změny kontextu, vzorkování zásobníku volání, sledování uspaných vláken, čtení hardwarových PMU nebo zobrazení programu v podobě jazyku symbolických adres. [TODO tracy]

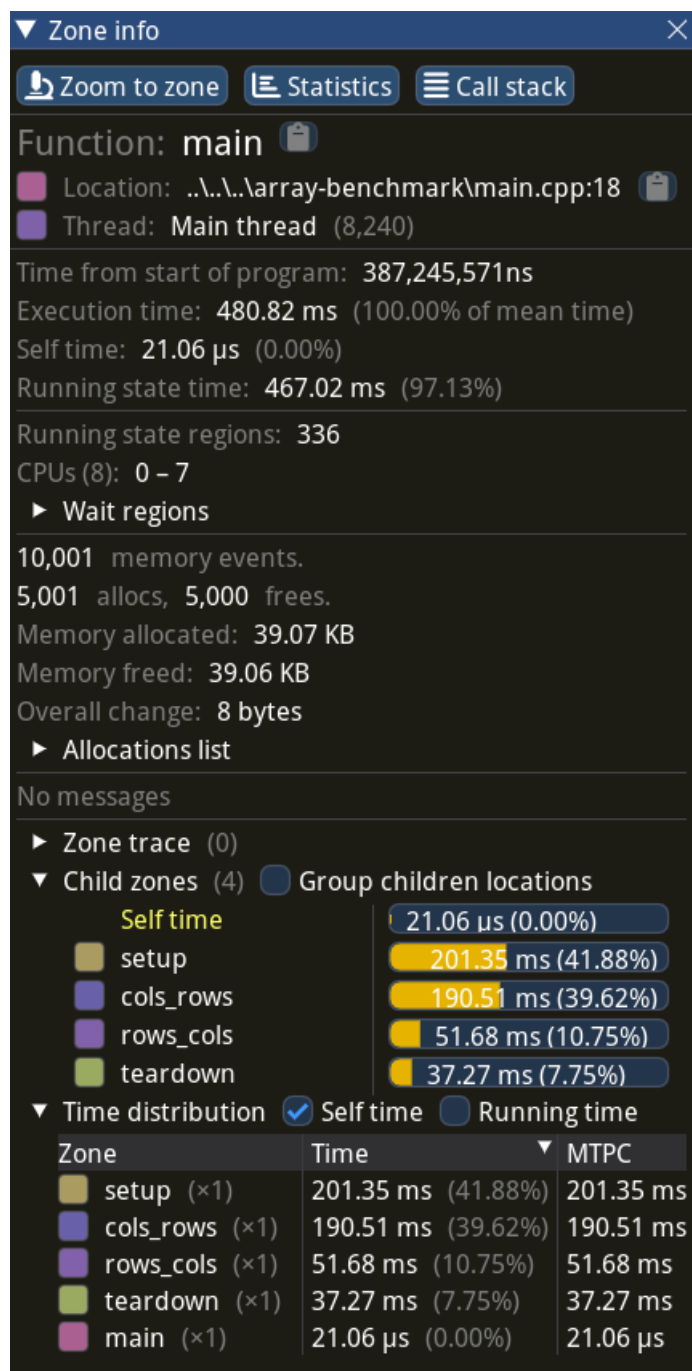
```
D:\Projekty\diplomka\Practice\libs\tracy\capture\build\win32\x64\Release>capture.exe -a 127.0.0.1 -o arrayTrace.tracy
Connecting to 127.0.0.1:8086...
Queue delay: 113 ns
Timer resolution: 12 ns
18.10 Mbps / 8.4% = 215.97 Mbps | Tx: 23.03 MB | 154.74 MB | 727.22 ms
Frames: 2
Time span: 727.22 ms
Zones: 5
Elapsed time: 13.53 s
Saving trace... done!
Trace size 62.74 MB (47.49% ratio)
```

Obrázek 30: Použití programu Tracy z příkazové řádky. Zdroj vlastní.



Obrázek 31: Grafické prostředí programu Tracy. Zdroj vlastní.

Grafické rozhraní programu Tracy nabízí velké množství funkcí a také získaných informací podaných v různé formě. Bezprostředně pod horní lištou se nachází timeline se zaznamenanými snímky. V tomto případě je zde pouze jeden, který obsahuje všechna data. Jelikož je program spuštěn s administrátorskými právy, vidíme rozšířené informace, jako například využití jednotlivých vláken a jader procesoru. Hlavní vlákno profilované aplikace je rozděleno na několik částí. Každá odpovídá funkci jazyka C++ a zastřešuje je funkce `main`. V grafu jsou také znázorněny časové okamžiky, ve kterých došlo ke čtení zásobníku volání. Pod tímto oknem ještě vidíme průběh využití CPU a operační paměti.



Obrázek 32: Informace o zóně. Zdroj vlastní.

Po kliknutí na část *main* v grafu jsou prezentovány zónové informace, které zobrazují dodatečná detailní data.

```
62     void rows_cols()  
63 @6 {  
64 @5     ZoneScopedS(5);  
65 @1     int sum = 0;  
66 @7     for (int i = 0; i < ROWS; i++)  
67         {  
68 @7         for (int j = 0; j < COLS; j++)  
69             {  
70 @9                 sum += arr[i][j];  
71 @1             }  
72 @1         }  
73 @8     }
```

Obrázek 33: Pohled na funkci ze zdrojového souboru. Zdroj vlastní.

Užitečná funkce tohoto programu spočívá ve sledování zásobníku volání a následné zobrazení samotného zápisu programu, který se vykonává. Jsou nabízeny dva pohledy. Jeden je načten ze zdrojového souboru a zobrazen, a druhý ve formě jazyka symbolických adres. Tento pohled mimo jiné také poskytuje informace o využití registrů a závislostech následujících instrukcí, nebo také informace o vlastnostech využitých instrukcí pro danou mikroarchitekturu. V případě použití podporovaného operačního systému dokáže program také zaznamenat informace z PMU a zobrazit je k odpovídající instrukci. Díky tomu můžeme přímo vidět hotspot v aplikaci a zaměřit se na optimalizace tam, kde dávají smysl. Rovněž dokážeme identifikovat závislosti mezi instrukcemi. Po kliknutí na vybranou instrukci se zobrazí názvy používaných registrů a také je naznačeno, do kterého se zapisuje a ze kterého se čte.


```

+0      push    rdi
+2      sub     rsp, 0x40
+6      lea     rdi, [rsp + 0x20]
+11     mov     ecx, 8
+16     mov     eax, 0cccccccc
+21     rep stosd dword ptr [rdi], eax
+23     mov     r9b, 1
+26     mov     r8d, 5
+32     lea     rdx, [rip + 0x40b71]
+39     lea     rcx, [rsp + 0x24]
+44     <call    0x7ff72dbc1d16
+49     mov     dword ptr [rsp + 0x34], 0
+57     mov     dword ptr [rsp + 0x38], 0
+65     jmp     .L1 -> [rows_cols]
+67     mov     eax, dword ptr [rsp + 0x38] ; .L0
+71     inc     eax
+73     mov     dword ptr [rsp + 0x38], eax
+77     cmp     dword ptr [rsp + 0x38], 0x1388 ; .L1
+85     jge     .L5 -> [rows_cols]
+87     mov     dword ptr [rsp + 0x3c], 0
+95     jmp     .L3 -> [rows_cols]
+97     mov     eax, dword ptr [rsp + 0x3c] ; .L2
+101    inc     eax
+103    mov     dword ptr [rsp + 0x3c], eax
+107    cmp     dword ptr [rsp + 0x3c], 0x1388 ; .L3
+115    jge     .L4 -> [rows_cols]
+117    movsxd  rax, dword ptr [rsp + 0x38]
+122    movsxd  rcx, dword ptr [rsp + 0x3c]
+127    mov     rdx, qword ptr [rip + 0x528ea]
+134    mov     rax, qword ptr [rdx + rax*8]
+138    mov     eax, dword ptr [rax + rcx*4]
+141    mov     ecx, dword ptr [rsp + 0x34]
+145    add     ecx, eax
+147    mov     eax, ecx
+149    mov     dword ptr [rsp + 0x34], eax
+153    jmp     .L2 -> [rows_cols]
+155    jmp     .L0 ; .L4 -> [rows_cols]
+157    lea     rcx, [rsp + 0x24] ; .L5
+162    <call    0x7ff72dbc1e56
+167    mov     rcx, rsp
+170    lea     rdx, [rip + 0x3fa9f]
+177    <call    0x7ff72dbc1a96
+182    add     rsp, 0x40
+186    pop     rdi
+187    <ret

```

Obrázek 34: Pohled na program v podobě jazyka symbolických adres. Zdroj vlastní.

II. PRAKTICKÁ ČÁST

5 POUŽITÉ TECHNOLOGIE

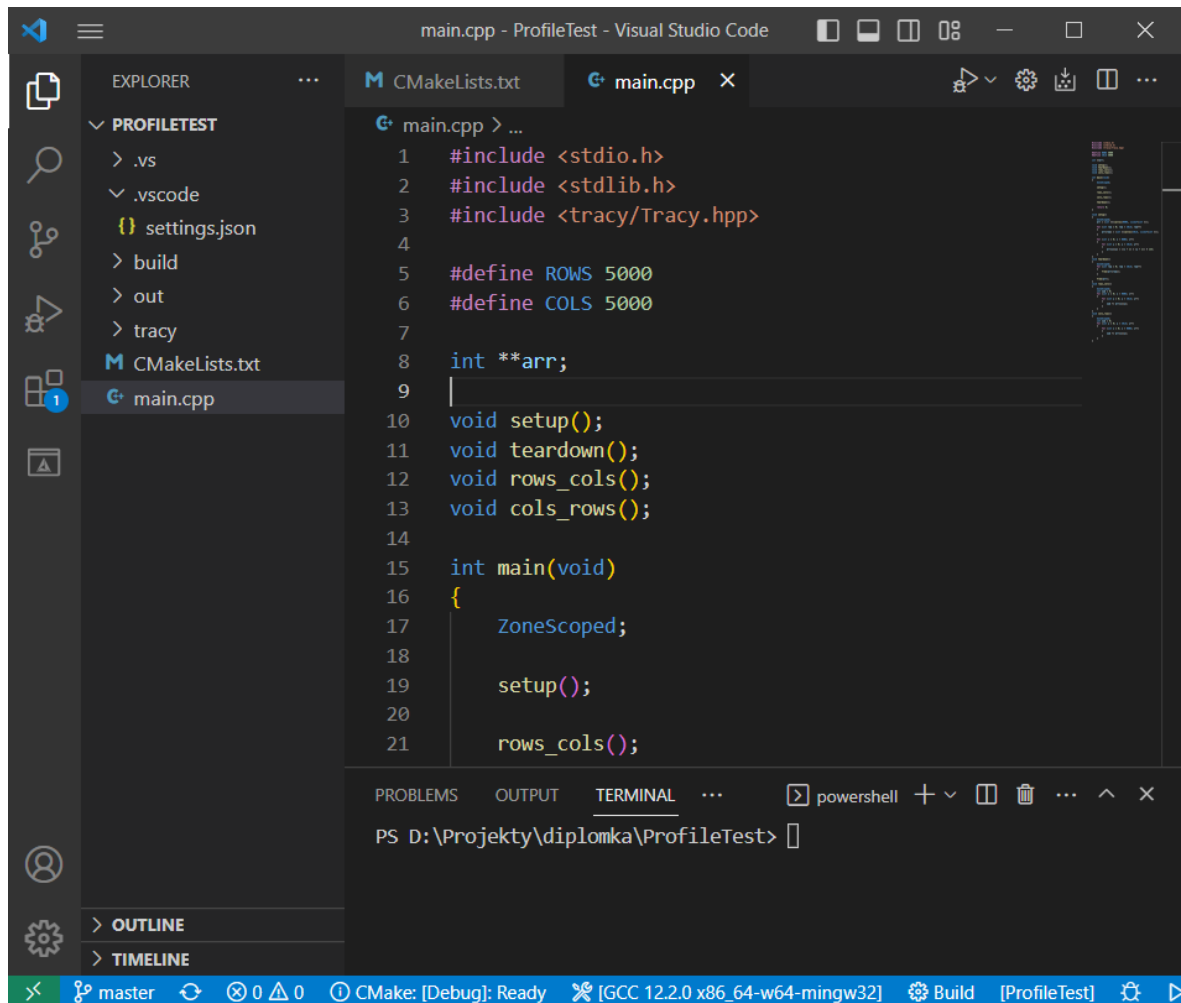
5.1 C++

„C++ je programovací jazyk pro obecné použití, který nabízí přímý a účinný hardwarový model a je vybaven pro definici odlehčených abstrakcí“. Tento jazyk může být využit k programování jak mikrokontrolerů, tak obrovských distribuovaných aplikací. Tvůrce tohoto jazyka zdůrazňuje jeden z principů, kterého se držel. Kromě C++ by se mezi něj a hardware neměl vejít další jazyk. Z toho důvodu je tento jazyk oblíbený pro programování systémů, protože dokáže přímo manipulovat s hardwarovými zdroji. Programátor zde může využít několik různých programovacích přístupů. *Procedurální* přístup se zaměřuje na zpracování a návrh vhodných datových struktur. Pro jeho využití slouží vestavěné datové typy jazyka, operátory, funkce a datové struktury. *Objektově orientovaný* přístup je podporován díky existenci tříd a možnostem dědičnosti a zapouzdření. *Obecný* přístup se soustředí na implementaci a použití obecných algoritmů. Pro zobecnění lze v C++ využít šablon. Pro vykonání programu napsaného v jazyku C++ se používá převod pomocí kompilace. Jeden z důvodů výběru tohoto jazyku pro diplomovou práci je existence více překladačů, čímž vzniká možnost porovnání vhodnosti každého z nich pro různé úkony. Tento jazyk je *staticky typovaný*, což znamená, že typ každé entity musí být známý kompilátorem v okamžiku používání. Datové typy odpovídají fundamentálním datovým typům využívaných počítačem. Každý datový typ má předem danou velikost, která je dána hardwarem. Patří mezi ně například *bool*, pro uchování pravdivostní hodnoty, *char*, pro zaznamenání znaku, *int*, pro kladná i záporná celá čísla, a *double*, pro čísla s desetinnou čárkou. Datové typy se využívají při *deklaraci* proměnných nebo funkcí. Deklarace představuje jméno do programu. Na rozdíl od běžné definice pojmu objekt, jazyk C++ jako *objekt* považuje kus paměti, která obsahuje hodnotu a má typ. *Proměnná* je poté pojmenovaný objekt. Pokud chceme s proměnnými pracovat, zpravidla k tomu využijeme *funkce*. Ta specifikuje, jak má být operace provedena. Při její deklaraci je třeba uvést návratový typ, jméno a případně argumenty této funkce, každý se svým datovým typem. Deklarovaná funkce sama o sobě nemá žádné chování. To je třeba specifikovat v *definici* funkce. [TODO c++]

5.2 Visual Studio Code

Jako primární vývojové prostředí bylo zvoleno Visual Studio Code. Jedná se o odlehčený a výkonný editor zdrojového kódu dostupný v podobě desktopové aplikace. Je dostupný pro

Windows, Linux a macOS. Obsahuje vestavěnou podporu jazyků JavaScript, TypeScript a runtime prostředí Node.js. Množství dalších jazyků je podporováno v podobě rozšíření. [TODO vscode]

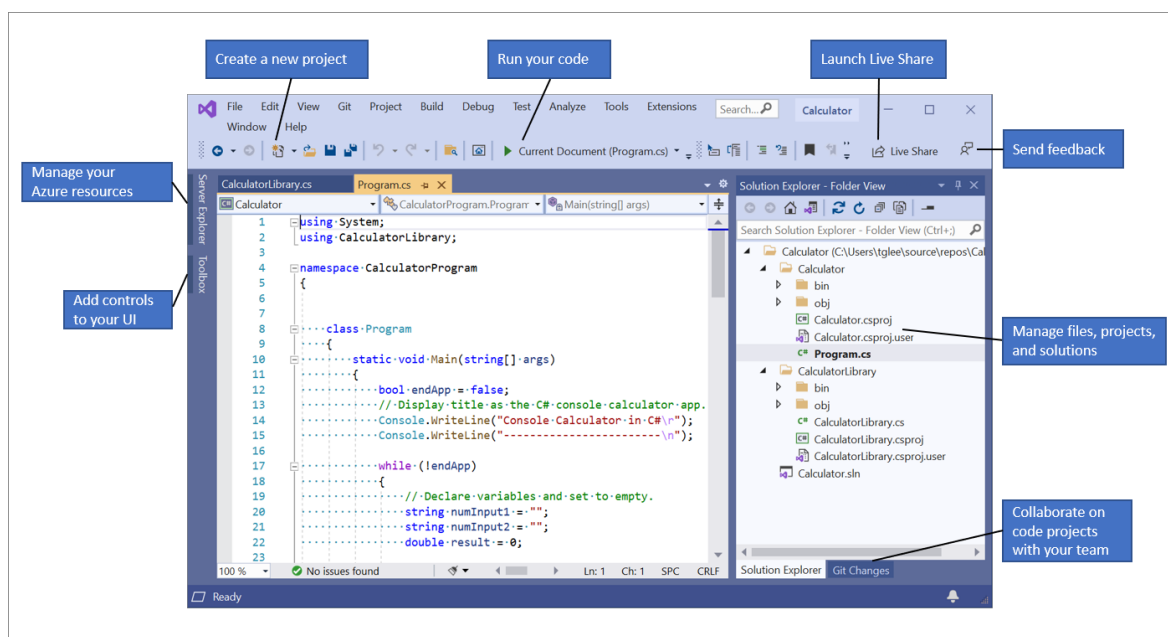


Obrázek 35: Grafické prostředí programu Visual Studio Code. Zdroj vlastní.

Tuto aplikaci lze vnímat zejména jako textový editor, který je posunut o několik úrovní výše díky volitelným rozšířením. Mimo ně se může pyšnit častými aktualizacemi, které jej obohacují o různé funkce zlepšující použitelnost. Aplikace se skládá z modulárních částí, jejichž pozici může uživatel měnit. Prostor disponuje prohlížečem souborů, hlavní plochou pro editaci textu, a také vestavěný terminál. Dále je grafické rozhraní obohaceno o ikony, tlačítka nebo okna podle nainstalovaných rozšíření. Toto můžeme vidět v obrázku v pravém horním rohu, kde se nachází ikony z rozšíření *Code Runner*. To slouží ke zjednodušení spouštění a ladění aplikace přímo z grafického rozhraní.

5.3 Visual Studio 2019

Dalším použitým vývojovým prostředím je Visual Studio 2019 ve verzi Community. Kromě editoru zdrojového kódu a nástrojů ladění toto vývojové prostředí nabízí také vlastní překladače, nástroje pro doplňování kódu, grafické návrháře a další nástroje pro ulehčení návrhu software. Jednou ze zajímavých funkcí je *Live Share*, která dovoluje více vývojářům kolaborovat na projektu a v reálném čase editovat a ladit kód. Vývojové prostředí je vybaveno dovednostmi navigace mezi zdrojovými soubory více různými způsoby, což programátorovi napomáhá při orientaci ve větších projektech. K usnadnění vývoje také přispívá technologie *IntelliSense*, kterou programátor využije při potřebě doplnit kód. Tento nástroj se snaží nabídnout nejrelevantnější výsledky na základě rozepsaného textu a také kontextu. Podobně jako předchozí vývojové prostředí, také Visual Studio 2019 nabízí množství možností úpravy vzhledu aplikace a rozložení oken. [TODO vs]



Obrázek 36: Vývojové prostředí Visual Studio 2019. [TODO vs]

5.4 CMake

V rámci praktické části je pro sestavování ukázkových aplikací využit systém CMake. Jedná se o open-source systém, který spravuje proces sestavování programu způsobem nezávislým na překladači. Pracuje s konfiguračním souborem s pevně daným názvem *CMakeLists*, který se nachází v každém zdrojovém adresáři. Díky těmto souborům dokáže generovat výsledné soubory. CMake dokáže podporovat složité adresářové hierarchie a také aplikace závislé na

více knihovnách. Díky jednoduchému návrhu tohoto systému je snadno rozšiřitelný o nové funkce. [TODO cmake-about]

V krátkém příkladu je nastíněna struktura konfiguračních souborů systému CMake. Uvažujeme hierarchii, ve které existuje jeden kořenový adresář, který obsahuje adresář *Hello* a adresář *Demo*. V kořenovém konfiguračním souboru specifikujeme minimální požadovanou verzi systému CMake, dále pak název projektu a specifikace podadresářů. Příklad dále ukazuje, jak vyjádříme, že adresář *Hello* je knihovna. Naproti tomu adresář *Demo* definuje spustitelný soubor s přilinkovanou knihovnou. [TODO cmake-examples]

```
# CMakeLists files in this project can
# refer to the root source directory of the project as ${HELLO_SOURCE_DIR} and
# to the root binary directory of the project as ${HELLO_BINARY_DIR}.
cmake_minimum_required (VERSION 2.8.11)
project (HELLO)

# Recurse into the "Hello" and "Demo" subdirectories. This does not actually
# cause another cmake executable to run. The same process will walk through
# the project's entire directory structure.
add_subdirectory (Hello)
add_subdirectory (Demo)
```

Obrázek 37: Konfigurační soubor kořenového adresáře. [TODO cmake-examples]

```
# Create a library called "Hello" which includes the source file "hello.cxx".
# The extension is already found. Any number of sources could be listed here.
add_library (Hello hello.cxx)

# Make sure the compiler can find include files for our Hello library
# when other libraries or executables link to Hello
target_include_directories (Hello PUBLIC ${CMAKE_CURRENT_SOURCE_DIR})
```

Obrázek 38: Konfigurační soubor knihovny Hello. [TODO cmake-examples]

```
# Add executable called "helloDemo" that is built from the source files
# "demo.cxx" and "demo_b.cxx". The extensions are automatically found.
add_executable (helloDemo demo.cxx demo_b.cxx)


# Link the executable to the Hello library. Since the Hello library has
# public include directories we will use those link directories when building
# helloDemo
target_link_libraries (helloDemo LINK_PUBLIC Hello)
```

Obrázek 39: Konfigurační soubor adresáře se spustitelným souborem. [TODO cmake-examples]

Podle konvence se výsledky sestavení zapisují do adresáře *build*. Tento adresář se vyskytuje ve stejné rovině, jako kořenový adresář. Jestliže se nacházíme v adresáři *build*, pak spustíme samotný proces sestavování pomocí volání `cmake ..` z příkazové řádky. V případě úspěšného sestavení jsou výsledné soubory vytvořeny v adresáři *build*.

5.5 Hardwarové specifikace

Následující programy budou spouštěny na počítači autora diplomové práce. Výsledky budou charakterizovat tuto konkrétní hardwarovou kombinaci. Jedná se o notebook Acer Aspire 5, konkrétně varianta A515-51G-55X7. Použitý operační systém je Microsoft Windows 10. Následující údaje byly pořízeny pomocí program CPU-Z.

Processor						
Name	Intel Core i5 8250U					
Code Name	Kaby Lake-R	Max TDP	15.0 W			
Package	Socket 1356 FCBGA					
Technology	14 nm	Core VID	1.043 V			
						
Specification	Intel® Core™ i5-8250U CPU @ 1.60GHz					
Family	6	Model	E	Stepping	A	
Ext. Family	6	Ext. Model	8E	Revision	Y0	
Instructions	MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, EM64T, VT-x, AES, AVX, AVX2, FMA3					
Caches (Core #0)						
Core Speed	3091.66 MHz					
Multiplier	x 31.0 (4.0 - 34.0)					
Bus Speed	99.73 MHz					
Rated FSB						
Cache						
L1 Data	4 x 32 KBytes		8-way			
L1 Inst.	4 x 32 KBytes		8-way			
Level 2	4 x 256 KBytes		4-way			
Level 3	6 MBytes		12-way			
Cores						
Selection	Socket #1		Cores	4	Threads	8

Obrázek 40: Informace o CPU. Zdroj vlastní.


General			
Type	DDR4	Channel #	Dual
Size	20 GBytes	DC Mode	
		Uncore Frequency	498.5 MHz
Timings			
DRAM Frequency	1197.1 MHz		
FSB:DRAM	3:36		
CAS# Latency (CL)	17.0 clocks		
RAS# to CAS# Delay (tRCD)	17 clocks		
RAS# Precharge (tRP)	17 clocks		
Cycle Time (tRAS)	39 clocks		
Row Refresh Cycle Time (tRFC)	420 clocks		
Command Rate (CR)	2T		
DRAM Idle Timer			
Total CAS# (tRDRAM)			
Row To Column (tRCD)			

Obrázek 41: Informace o operační paměti. Zdroj vlastní.

Display Device Selection

Intel(R) UHD Graphics 620 Perf Level Current

GPU

Name	Intel® UHD Graphics 620			
Board Manuf.	ACER			
Code Name		Revision	7	
Technology		TDP		

Clocks

GFX Core	
Shader / SoC	
Memory	

Memory


Size	
Type	
Vendor	
Bus Width	

Obrázek 42: Informace o grafické kartě #1. Zdroj vlastní.

Display Device Selection

NVIDIA GeForce MX150 Perf Level Current

GPU

Name	NVIDIA GeForce MX150			
Board Manuf.	ACER			
Code Name	GP 108	Revision	A1	
Technology	14 nm	TDP	25.0 W	

Clocks

GFX Core	607.5 MHz
Shader / SoC	
Memory	405.0 MHz

Memory

Size	2 GBytes
Type	GDDR5
Vendor	Hynix
Bus Width	64 bits

Obrázek 43: Informace o grafické kartě #2. Zdroj vlastní.

6 PŘÍKLADY OPTIMALIZACÍ

V následující sekci budou přestaveny vzorové programy, které znázorňují jeden nebo více principů nastíněných v teoretické části. Představené úpravy kódu mohou mít různý vliv na výkon v závislosti na platformě a na konkrétní aplikaci pracující s daty. Dosažené výsledky budou kvantifikovány a zhodnoceny v následující sekci.

6.1 Iterace polem

Většina programovacích jazyků nabízí mechanismus, který dovoluje opakovaně provádět stejnou posloupnost operací nad daty. V jazyce C++ k tomu slouží konstrukty jako *for*, *while* či *do-while*. Při využití smyčky *for* může programátor provést potřebná nastavení v inicializační části, stanovit ukončovací podmínku a také specifikovat velikost kroku od iterace k iteraci. Běžně je pro realizaci opakovaného vykonávání využito instrukce podmíněného skoku. Nejprve se provede vyhodnocení ukončovací podmínky a poté buď pokračujeme do další iterace, nebo smyčka končí.

6.1.1 Iterace dvourozměrným polem

Velmi jednoduchá ukázka spolupráce s charakteristikami hardware jsou odlišné způsoby iterace přes dvourozměrné pole. Jeden z nich bude vždy rychlejší i přesto, že jejich asymptotická složitost je totožná. Rychlejší způsob využívá faktu, že „prvky pole jsou uspořádány v paměti po řádcích.“ [TODO csprogrammer] Dále je třeba si uvědomit, že při čtení i jednoho bytu nám z operační paměti dorazí celý cache blok. V případě, že jsou sousedící prvky ve vyrovnávací paměti, CPU k nim může přistoupit rychleji.


```
1 #define ROWS 5000
2 #define COLS 5000
3
4     int **array;
5
6     // Array initialization
7
8     int sum = 0;
9     for (int i = 0; i < ROWS; i++)
10    {
11        for (int j = 0; j < COLS; j++)
12        {
13            sum += array[i][j];
14        }
15    }
16
17    sum = 0;
18    for (int j = 0; j < COLS; j++)
19    {
20        for (int i = 0; i < ROWS; i++)
21        {
22            sum += array[i][j];
23        }
24    }
```

Zdrojový kód 1: Průchod dvourozměrným polem.

Tento princip lokality je možné exploatovat při násobení matic. Matice mohou být iterovány v různém pořadí, a toto pořadí může mít vliv na rychlost běhu programu.

```
1     double **A;
2     double **B;
3     double **C;
4
5     double sum = 0.0;
6     int n = // A number large enough to ensure one row does not fit
7 the L1 cache
8 // Version 1
9     for (int i = 0; i < n; i++)
10    {
11        for (int j = 0; j < n; j++)
12        {
13            sum = 0.0;
14
15            for (int k = 0; k < n; k++)
16            {
17                sum += A[i][k] * B[k][j];
18            }
19
20            C[i][j] = sum;
21        }
22    }
23
24    // Version 2
25    for (int j = 0; j < n; j++)
```

```
26     {
27         for (int i = 0; i < n; i++)
28         {
29             sum = 0.0;
30
31             for (int k = 0; k < n; k++)
32             {
33                 sum += A[i][k] * B[k][j];
34             }
35
36             C[i][j] = sum;
37         }
38     }
39
40     // Version 3
41     double r = 0.0;
42     for (int j = 0; j < n; j++)
43     {
44         for (int k = 0; k < n; k++)
45         {
46             r = B[k][j];
47             for (int i = 0; i < n; i++)
48             {
49                 C[i][j] += A[i][k] * r;
50             }
51         }
52     }
53
54     // Version 4
55     for (int k = 0; k < n; k++)
56     {
57         for (int j = 0; j < n; j++)
58         {
59             r = B[k][j];
60             for (int i = 0; i < n; i++)
61             {
62                 C[i][j] += A[i][k] * r;
63             }
64         }
65     }
66
67     // Version 5
68     for (int k = 0; k < n; k++)
69     {
70         for (int i = 0; i < n; i++)
71         {
72             r = A[i][k];
73             for (int j = 0; j < n; j++)
74             {
75                 C[i][j] += r * B[k][j];
76             }
77         }
78     }
79
80     // Version 6
81     for (int i = 0; i < n; i++)
```

```
82     {  
83         for (int k = 0; k < n; k++)  
84         {  
85             r = A[i][k];  
86             for (int j = 0; j < n; j++)  
87             {  
88                 C[i][j] += r * B[k][j];  
89             }  
90         }  
    }
```

Zdrojový kód 2: Různé způsoby průchodu maticí při sčítání matic. [TODO csprogrammer]

6.1.2 Závislost iterací [TODO caqa str. 346]

6.1.3 Loop unrolling

6.1.4 Projevy asociativity vyrovnávacích paměti [TODO cache-effects]

6.2 Předvídatelnost operací

6.3 Hot vs cold data

6.4 SoA vs AoS

6.5 Využití pipeliningu

6.6 Zarovnání dat

6.7 Datové a kontrolní závislosti

6.8 Datové sdílení a falešné sdílení [TODO caqa str. 396]

6.9 Aliasing paměti [TODO csprogrammer str. 536]

6.10 Návrhové vzory z pohledu DOP

7 OVĚŘENÍ VÝKONU POMOCÍ NÁSTROJŮ

8 SOUHRN OPTIMALIZACÍ

ZÁVĚR

text

SEZNAM POUŽITÉ LITERATURY

[1] Text

TODO – očíslovat a smazat

Acton2014 – CppCon 2014: Mike Acton "Data-Oriented Design and C++". *Youtube* [online]. 30.9.2014 [cit. 2022-12-21]. Dostupné z:

<https://www.youtube.com/watch?v=rX0ItVEVjHc>. Kanál uživatele CppCon

dodmain - FABIAN, Richard. Data-Oriented Design: Software engineering for limited resources and short schedules [online]. Richard Fabian, 2018, 307 s. ISBN 9781916478701.

llopis – LLOPIS, Noel. Data-Oriented Design (Or Why You Might Be Shooting Yourself in The Foot With OOP) [online]. 2009-12-04 [cit. 2022-12-23]. Dostupné z: <https://gamesfromwithin.com/data-oriented-design>

mozillaOOP - Object-oriented programming [online]. 2022-09-28 [cit. 2022-12-30]. Dostupné z: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object-oriented_programming

wtfp - MITCHELL, Brad. What Is Functional Programming and Why Use It? [online]. 2022-07-13 [cit. 2022-12-30]. Dostupné z: <https://www.codingdojo.com/blog/what-is-functional-programming>

csprogrammer – BRYANT, Randal a David O'HALLARON. Computer Systems: A Programmer's Perspective. 3rd Edition. Pearson, 2015, 1128 s. ISBN 013409266X.

mutluPrefetch – Lecture 25: Prefetching - Carnegie Mellon - Computer Architecture 2015 - Onur Mutlu. *Youtube* [online]. 30.9.2014 [cit. 2022-12-30]. Dostupné z: <https://www.youtube.com/watch?v=ibPL7T9iEwY>. Kanál uživatele Carnegie Mellon Computer Architecture.

mutluPrefetch2 – Lecture 26. More Prefetching and Emerging Memory Technologies - CMU - Comp. Arch. 2015 - Onur Mutlu. *Youtube* [online]. 30.9.2014 [cit. 2022-12-30]. Dostupné z: <https://www.youtube.com/watch?v=TUFins4z6o4>. Kanál uživatele Carnegie Mellon Computer Architecture.

caqa – HENNESSY, John L. a David A. PATTERSON. Computer Architecture: A Quantitative Approach. 4th Edition. 2006, 704 s. ISBN 0123704901.

asm - KUSSWURM, Daniel. Modern X86 Assembly Language Programming: 32-bit, 64-bit, SSE, and AVX. Apress, 2014, 700 s. ISBN 1484200659.

gcc – STALLMAN, Richard M. Using the GNU Compiler Collection: For gcc version 12.2.0 [online]. GNU Press, 2022 [cit. 2023-01-16]. Dostupné z: <https://gcc.gnu.org/onlinedocs/gcc-12.2.0/gcc.pdf>

clang – Clang Compiler User's Manual [online]. [cit. 2023-01-16]. Dostupné z: <https://clang.llvm.org/docs/UsersManual.html>

msvcOptions - Compiler Options [online]. [cit. 2023-01-16]. Dostupné z: <https://learn.microsoft.com/en-us/cpp/build/reference/compiler-options>

intrinsics - Intel® Intrinsics Guide [online]. 2022-12-14 [cit. 2023-01-27]. Dostupné z: <https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html>

micro - What is Micro-benchmarking? [online]. 2022-01-14 [cit. 2023-02-04]. Dostupné z: <https://www.adservio.fr/post/what-is-microbenchmarking>

profile – The Basics of Profiling - Mathieu Ropert - CppCon 2021. *Youtube* [online]. 18.2.2022 [cit. 2022-02-05]. Dostupné z: <https://www.youtube.com/watch?v=dToaepIXW4s>. Kanál uživatele CppCon

vtune-guide – Intel® VTune™ Profiler User Guide [online]. 2022-12-16 [cit. 2023-02-06]. Dostupné z: <https://www.intel.com/content/www/us/en/develop/documentation/vtune-help/top.html>

vtune-cook – Intel® VTune™ Profiler Performance Analysis Cookbook [online]. 2022-12-16 [cit. 2023-02-06]. Dostupné z: <https://www.intel.com/content/www/us/en/develop/documentation/vtune-cookbook/top.html>

tracy – TAUDUL, Bartosz. Tracy Profiler: The user manual [online]. 2022-10-26 [cit. 2023-02-06]. Dostupné z: <https://github.com/wolfpld/tracy/releases/latest/download/tracy.pdf>

c++ - STROUSTRUP, Bjarne. The C++ Programming Language. 4th Edition. Addison-Wesley Professional, 2013, 1376 s. ISBN 0275967301.

vscode - Getting Started. Visual Studio Code [online]. [cit. 2023-02-18]. Dostupné z: <https://code.visualstudio.com/docs>

vs – Welcome to the Visual Studio IDE [online]. 2022-09-21 [cit. 2023-02-19]. Dostupné z: <https://learn.microsoft.com/en-us/visualstudio/get-started/visual-studio-ide?view=vs-2019>

cmake-about – About CMake [online]. [cit. 2023-02-19]. Dostupné z: <https://cmake.org/overview/>

cmake-examples – Examples [online]. [cit. 2023-02-19]. Dostupné z: <https://cmake.org/examples>

cache-effects - CPU Cache Effects - Sergey Slotin - Meeting C++ 2022. *Youtube* [online]. 6.12.2022 [cit. 2023-02-27]. Dostupné z:

https://www.youtube.com/watch?v=mQWuX_KgH00. Kanál uživatele Meeting Cpp

TODO – zkontrolovat na <https://odevzdej.cz/>

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

DOD	Data-oriented design
DOP	Datově orientované programování
FP	Funkcionální programování
OOP	Objektově orientované programování
SSD	Solid State Drive
SRAM	Static Random Access Memory
CPU	Central Processing Unit
LRU	Least Recently Used
LFU	Least Frequently Used
DRAM	Dynamic Random Access Memory
ISA	Instruction Set Architecture
SIMD	Single Instruction Multiple Data
MMX	Multi Media Extensions
SSE	Streaming SIMD Extensions
AVX	Advanced Vector Extensions
CPI	Cycles Per Instruction
IPC	Instructions Per Cycle
GPU	Graphics Processing Unit
PMU	Performance Monitoring Unit

SEZNAM OBRÁZKŮ

Obrázek 1. Ukázkový obrázek.....13

Obrázek 3: Příklad rozdělení adresy.....13

TODO

SEZNAM TABULEK

Tabulka 1. Ukázková tabulka	13
-----------------------------------	----

SEZNAM ZDROJOVÝCH KÓDŮ

TODO

SEZNAM PŘÍLOH

PŘÍLOHA P I: NÁZEV PŘÍLOHY