|  |  |
| --- | --- |
|  | |
| **Datově orientovaný přístup při vývoji software** | |
|  | |
| Bc. Tomáš Janečka | |
|  | |
|  |  |
| 2023 | Popis: fai_logo_cz |
|  |  |
|  | |

Text, letter

Description automatically generated

Text, letter

Description automatically generated**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, memoryTODO

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

OBSAH

[Úvod 8](#_Toc120888191)

1. [TEORETICKÁ ČÁST 9](#_Toc120888192)

[1 Nadpis Error! Bookmark not defined.](#_Toc120888193)

[1.1 Podnadpis **Error! Bookmark not defined.**](#_Toc120888194)

[1.2 Podnadpis **Error! Bookmark not defined.**](#_Toc120888195)

[1.2.1 Podpodnadpis **Error! Bookmark not defined.**](#_Toc120888196)

[2 Nadpis Error! Bookmark not defined.](#_Toc120888197)

[2.1 Podnadpis **Error! Bookmark not defined.**](#_Toc120888198)

[2.1.1 Podpodnadpis **Error! Bookmark not defined.**](#_Toc120888199)

1. [Praktická část 55](#_Toc120888200)

[3 Nadpis Error! Bookmark not defined.](#_Toc120888201)

[3.1 Podnadpis **Error! Bookmark not defined.**](#_Toc120888202)

[3.2 Podnadpis **Error! Bookmark not defined.**](#_Toc120888203)

[4 Nadpis Error! Bookmark not defined.](#_Toc120888204)

[4.1 Podnadpis **Error! Bookmark not defined.**](#_Toc120888205)

[Závěr 56](#_Toc120888206)

[Seznam použité literatury 85](#_Toc120888207)

[Seznam použitých symbolů a zkratek 88](#_Toc120888208)

[Seznam obrázků 89](#_Toc120888209)

[Seznam tabulek 90](#_Toc120888210)

[Seznam Příloh 92](#_Toc120888211)

TODO – obsah poskládat manuálně

Úvod

Objektově orientované programování je jedno z nejrozšířenějších paradigmat mezi programátory. Mnoho z nich, včetně autora, 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. Zároveň se toto paradigma soustředí na charakteristiky hardware, na kterém software běží a dbá na efektivní využívání zdrojů. Dostupnost výpočetních zdrojů je v porovnáním s minulostí nesrovnatelná, což ale neznamená, že je pro určité aplikace dostačující. 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 poukazují 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. Autor tyto nápady považuje za velmi zajímavé, a proto tato diplomová práce pojednávat o shrnutí nejzásadnějších myšlenek tohoto paradigmata. V teoretické části bude představena definice DOP a budou popsány 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. Na základě empirických dat bude sestavena sada doporučení pro čtenáře, kteří by se chtěli zdokonalit v této oblasti a využít benefitů, které psaní programů tímto způsobem přináší.

|  |  |
| --- | --- |
|  | TEORETICKÁ ČÁST |

# Datově orientované programování

V této sekci bude představen 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 omezeného 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.

## 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.“ [1]

„Datově orientovaný návrh si nechává napovědět daty, která jsou pozorovatelná nebo očekávaná. Na rozdíl od uvažování všech možných scénářů nebo plánování adaptability, využíváme nejpravděpodobnější vstupy pro nasměrování algoritmu. Na rozdíl od plánování rozšiřitelnosti je jednoduchý a má za cíl splnit svůj úkol.“ [1]

## 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, jehož autorem je Noel Llopis. 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 článku 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 démonizovat. Například „v systémech, které byly navrženy tímto způsobem nebo výkonově nekritické aplikace.“ [2]

Odkazy z tohoto příspěvku směřují na Mika Actona a Jima Tilandera. Oba jsou vlastníky webů, kde v minulosti publikovali blogy na různá témata, které zjevně ovlivnily Llopisův 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.

## Hlavní myšlenky

### 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 třeba myslet na to, že data nikdy neexistují jen tak, ale pokaždé se nachází na nějakém hardware, ať už na virtuálním stroji, nebo konkrétním procesoru. [1]

### 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 se v tomto paradigmatu uvažuje o datech jako o „faktech, o kterých můžeme uvažovat tak, jak potřebujeme pro získání výstupních dat v požadovaném formátu.“ [1]

### Statistika

„Data jsou typ, frekvence, množství, tvar a pravděpodobnost.“ [1]

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 se píše kód. Pokud je k dispozici prvotní verzi funkční aplikace, která pracuje s produkčními daty, nebo disponujeme daty, která byla shromážděna libovolným způsobem, má programátor více kontextu a dokáže lépe uvažovat o problému, který řeší.

„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. [3]

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.

### 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 dokáže programátor lépe reagovat na změny a minimalizovat náklady potřebného přepisu kódu. [1]

## Využití

Jak již bylo zmíněno, datově orientovaný návrh je hojně využíván 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 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 se porozumí datům, porozumí se problému a programátor může 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 vypozorování vzorce opakování hodnot proměnných.

# další paradigmata

Programovací paradigma je způsob nebo styl, kterým se píše 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 paradigmata, 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.

## 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.“ V rámci tohoto paradigmata 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. [4]

Pokud rodičovská třída definuje metody jako virtuální, pak třídy, které od ní dědí, mohou měnit chování v závislosti na typu instance. Toto 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. [4]

## 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.“ 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. [5]

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.

## 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 řešeném problému.“ [1]

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, nehledě 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 překopíruje 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. [5]

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 paradigmata 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 modularita aplikace, datově orientovaný přístup se soustředí jak na modularitu, tak na zvýšení výkonu programu.

# hardware jako platforma

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

## 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 mluví o cache pamětech, myslí se zpravidla uložiště, které se nachází na procesorovém čipu a je k dispozici výpočetním jádrům. Jako vyrovnávací paměť se ale může považovat i operační paměť ve vztahu k hard-disku nebo SSD. [6]

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. [6]

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. [7]

Chart, line chart, scatter chart

Description automatically generated

Obr. : Vývoj doby trvání přístupu do procesorové a operační paměti. [7]

### Struktura a adresování

Cache paměti uspořádávají data do sad (set). Každá sada obsahuje jeden nebo více řádků (line). Jeden řádek obsahuje kromě samotných dat, kterým se také říká blok (block), obsahuje ještě jeden bit *v*,který signalizuje validitu záznamu, a také štítek (tag), které se používají při adresování a načítání dat z cache. [6]

K adresování do vyrovnávací paměti se využívá celá adresa dat rozdělená na části. Počet bitů každé části je dán počtem bitů, které počítač používá pro adresování *m* (typicky 32 nebo 64 bitů), počtem sad *S* a velikostí bloku *B* (typicky 64 B). Parametry *S*a *B* jsou stanoveny výrobcem a každá konfigurace má své výhody i nevýhody. Abychom mohli získat počet bitů použitých pro tag *t*, vypočítáme počet bitů pro index sady a počet bitů pro offset v bloku . Počet bitů pro tag . [6]

A picture containing background pattern

Description automatically generated

Obr. : Příklad rozdělení adresy. [Zdroj vlastní]

Bity *s* slouží pro nalezení odpovídající sady v cache paměti. Bity *t* slouží pro identifikaci odpovídající cache line v sadě. Bity *b* říkají, kde v rámci je bloku začátek dat, o která se žádá. [6]

### 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 . 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. [6]

### 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*. [6]

### Set associative vyrovnávací paměť

Tento typ pamětí se redukuje problém existující v paměti s přímým mapováním pomocí navýčení 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. [6]

Při návrhu stupně asociativity je třeba zvolit vhodný kompromis mezi větším a nižším. Vyšší stupeň asociativity je pomalejší a složitější na implementaci. [6]

Table

Description automatically generated

Obr. : Příklad uspořádání a typu cache pamětí – snímek programu CPU-Z. [Zdroj vlastní]

Obr. 3 je příkladem různých úrovní cache pamětí. 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.

### 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. [6]

### Substituční strategie

Pokud nastane *conflict miss*, je třeba umístit žádaná data na vhodné místo. Toto místo může být zvoleno náhodně. Alternativně je možné zvolit sofistikovanější postupu, jako je nahrazení dat, která byla použita nejdále v minulosti (LRU), případně nejméně často využita ve stanoveném časovém okně (LFU). [6]

### Způsoby zapisování

Na rozdíl od čtení dat, zápis dat je o něco složitější. Jsou-li data, která jsou aktualizována, v cache paměti, mohou být nová data buď rovnou zapsána do cache paměti nižší úrovně (*write-through*) nebo upravena a zapsána je do paměti až v době vyřazení cache line z paměti (*write-back*). Nastane-li *write miss*, může se opět vybrat z více přístupů. 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. [6]

## 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. [6]

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 zapsání dat z požadované paměťové lokace z bufferu řádku na datové vodiče. [6]

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. [6]

## CPU pipelining

### 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. Aby mohla být hodnocena vhodnost různých přístupů k provádění instrukcí, jsou využívány pojmy *latency*, což je doba vykonání operace, a *throughput*, který popisuje počet provedených operací za jednotku času. [6]

### Sekvenční zpracování

Jestliže se vykonává instrukce jedna po druhé a každá instrukce projde kanálem jako celek, hovoří se o sekvenčním vykonávání.

Diagram, table

Description automatically generated

Obr. : Sekvenční vykonávání instrukcí. [Zdroj vlastní]

### Skalární pipeline

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

Table

Description automatically generated with medium confidence

Obr. : Skalární instrukční pipeline. [Zdroj vlastní]

### 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.

Table

Description automatically generated

Obr. : Superskalární instrukční pipeline. [Zdroj vlastní]

### 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 snížení doby trvání každé fáze, což, jako každé rozhodnutí při návrhu hardware, má taky své nevýhody. [6]

### Závislosti

Doposud byla uvažována nezávislost dvou po sobě jdoucích instrukcí. V reálném světě k takovému ideálnímu případu nemusí docházet často. Rozlišuje se *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 výpočtech, které bývají označovány jako *data* a *control hazard*. [6]

Table

Description automatically generated

Obr. : Data hazard. [Zdroj vlastní]

*Data hazard* může mít podobu načtení staré hodnoty aktuální instrukcí, zatímco předchozí instrukce už měla hodnotu změnit na novou. Obr. 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. [6]

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. [6]

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. Obr. 7 také 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. [7]

## 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. [8]

### 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. [8]

### 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*). [8]

### Execution based prefetching

Univerzální způsob, který může být implementován jak hardwarově, tak softwarově, je *execution based prefetching*. Zde probíhá přednačítání dat v jiném vlákně, než ve kterém je prováděn hlavní program. [8]

Ve speciálním vlákně 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. [9]

### 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 vypozorovat, 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*. [9]

### Content directed prefetching

Při programování v jazycích jako C/C++ je často využíváno práce s ukazateli do paměti, zejména pak u některých datových struktur. Jelikož ukazatel se může odkazovat na jakékoliv místo v paměti, jen těžko bych mohl programátor 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. [9]

## 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 zužitkování lokality. Lze 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 označování 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. [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. [6]

### 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*. [10]

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. Obr. 8 ukazuje grafické znázornění. [10]

Table

Description automatically generated

Obr. : Vnitřní architektura x86-32. [10]

### 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 použit libovolné kombinace těchto hodnot. [10]

### 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. [10]

### 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*. [10]

### 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. [10]

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. [6]

### Instrukce pro předání kontroly

Instrukce *jmp* provede skok na specifikované návěstí. 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. Obr. 9 ukazuje příklad programu v jazyce symbolických adres. [10]

Text

Description automatically generated

Obr. : Příklad programu v jazyce symbolických adres. [10]

## Překladače

Překlad z high-level jazyka do strojového kódu se odehrává v několika krocích, které bývají označovány jako *kompilační systém*. V první částí 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. [6]

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

### GCC

Zkratka GCC znamená *GNU Compiler Collection* a jedná se o integrovanou distribuci překladačů pro programovací jazyky jako je C, C++ či Fortran. [11]

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



Obr. : 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*. [11]

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 *-fsso-struct*, za který lze dosadit *big-endian*, *little-endian* nebo *native*. [11]

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. [11]

### 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.“ [12]



Obr. : 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. [12]

### 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. [13]



Obr. : Příklad běžné kompilace programu pomocí MSVC. [Zdroj vlastní]

## 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. [10]

Table

Description automatically generated

Obr. : Možnosti využití registru v SIMD. [10]

Obr. 13 znázorňuje možnosti rozdělení SIMD registru. 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 lze 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. [10]

Table

Description automatically generated

Obr. : Sada registrů AVX. [10]

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. [10]

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. Jsou 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*). [10]

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

Text, letter

Description automatically generated

Obr. : 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 lze pomocí adekvátních přepínačů u překladače GCC docílit využití operací nad složenými typy.

Text

Description automatically generated

Obr. : Kód varianty s přepínači *-O1 -msse*. [Zdroj vlastní]

A screenshot of a computer

Description automatically generated with low confidence

Obr. : Kód varianty s přepínači *-O2 -msse*. [Zdroj vlastní]

Jak je zřejmé z Obr. 16 , 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.

Text

Description automatically generated

Obr. : 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). [14]

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.

## Paralelizace

Paralelizace ve smyslu provádění výpočtů na více jádrech procesoru v jeden okamžik je v současné době hojně využívaná 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.

### 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 softwarově, 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. [7]

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*. [6]

Diagram, table

Description automatically generated

Obr. : Exekuční engine Haswell CPU. [10]

Z důvodu vícenásobného počtu exekučních jednotek, znázorněných v Obr. 19 pro tu samou operaci můžeme docílit souběžného vykonávání více mikro-operací stejného typu.

#### 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. [7]

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 otevírá dveře pro aplikaci dalších optimalizací, jako například přeskládání výrazů. [7]

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 výt využit. Z důvodu zachování nezávislosti instrukcí musí být mezivýsledky ukládány do odlišných registrů, jejichž počet je omezený. [7]

#### 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. [7]

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*. Této 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. [6]

### Data-level paralelismus

Mluví-li se o paralelismu na úrovni dat, myslí se tím především SIMD výpočetní model. Mohou sem být zařazeny 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. [7]

#### 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. [7]

Pokud je známa maximální povolenou délku registru pro vektorové operace, mohou být data a smyčky nachystány tak, aby bylo data možné na registry namapovat. Takový postup je označován jako *strip mining*. Pokud se v původním programu vyskytovala jedna smyčka, bude rozdělena na dvě. Hlavní smyčka zpracuje data, kterými lze plně naplnit vektorové registry a vedlejší smyčka, která zpracuje zbytek. [7]

#### 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. [7]

#### Loop-level paralelismus

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 závislost popisuje využití dat vypočítaných v předchozích iteracích iteracemi následujícími. [7]

A picture containing text, antenna

Description automatically generated

Obr. : Ukázka *loop-carried* závislosti. [7]

Příklad v Obr. 20 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. [7]

Text

Description automatically generated

Obr. : Přepis smyčky. [7]

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. 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. [7]

### 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*. [7]

#### 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. [7]

Zrychlení je vyjádřeno jako:

|  |  |
| --- | --- |
|  |  |

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

|  |  |
| --- | --- |
|  |  |

* je doba běhu algoritmu s vylepšením
* je doba běhu algoritmu bez vylepšení

Celkové zrychlení je vyjádřeno jako:

|  |  |
| --- | --- |
|  |  |

Tento zákon vyjadřuje klesající výtěžnost implementace nových vylepšení. Velikost části programu, na kterou může být aplikováno vylepšení, diktuje maximální možné dosažitelné zrychlení. [7]

#### Koherence vyrovnávacích pamětí

U *multicore* systému můžeme pozorovat asymetrický přístup do paměti. Načítání dat z vyrovnávací paměti je rychlejší než načítaní 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 takových 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. [7]

#### 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. [7]



Obr. : Příklad atomické výměny hodnot. [7]

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. [7]

Graphical user interface, text, application

Description automatically generated

Obr. : Spin lock. [7]

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. [7]

#### 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. [6]

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. [6]

POSIX vlákna poskytují standardní rozhraní pro práci s vlákny v jazyce C. Funkce pro vytvoření vláka 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 vláka. Všechna existující vlákna mohou být zrušena voláním funkce pthread\_exit a pro jednotlivá vlákna slouží pthread\_cancel. Jestliže se od vlákna očekává návratová hodnoa, po dokončení jeho běhu je obdržen výsledek pomocí pthread\_join, které se předá identifikátor vlákna a ukazatel pro data. [6]

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 nelze 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. Těmto skupinám se říká *kritická sekce* a můžeme si ji zobrazit v *grafu průběhu* v Obr. 24. [6]

Diagram

Description automatically generated

Obr. : Graf průběhu. [6]

#### 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 se volá 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, mluví se o *binárním semaforu*, či *mutexu*. Využití takového synchronizačního objektu můžeme promítnout do grafu průběhu v Obr. 25. [6]

Chart, box and whisker chart

Description automatically generated

Obr. : Graf průběhu s promítnutými hodnotami binárního semaforu. [6]

Tuto synchronizační strukturu lze také využít pro signalizaci dostupnosti sdílených prostředků. Toho se docílí inicializací semaforu na hodnotu větší než 1. Toto lze ilustrovat na příkladu *producent-konzument*, kde mimo výlučný přístup ke sdílenému datovému bufferu je třeba také zajistit uspořádaný přístup. Proto se zde ještě musí použít další dva semafory. 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. [6]

# měření výkonu aplikací

Bude-li se 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. [7]

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. Je-li znám počet cyklů pro provedení celého programu, zjistí se *procesorový čas*. [7]

|  |  |
| --- | --- |
|  |  |

V případě, že je znám počet instrukcí v programu, zjistí se průměrný počet cyklů na instrukci, označováno jako CPI. [7]

|  |  |
| --- | --- |
|  |  |

## Benchmarking

Návrháři výpočetního hardware využívají zátěžové testy, zvané *benchmark*, 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. [7]

### Microbenchmark

Microbenchmark měří výkon malé části programu. Nemohou být použity pro posouzení celkového výkonu aplikace, ale jsou vhodné pro posouzení latence operací a propustnosti. [15]

V rámci této práce bude využita knihovna Google Benchmark, která slouží k ohodnocení výkonu funkcí v programu, podobně jako u jednotkových testů. Jedná se o rozšířenou knihovnu pro provádění měření pro demonstraci výkonu ukázkových je dostačující.

## Profilování

Optimalizace se mimo jiné zabývá snižováním doby běhu programu. Aby se dalo zjistit, proč program neběží tak rychle, jak by mohl, je využíváno profilování. K tomu slouží nástroje zvané *profilery*, které umožňují měřit určité metriky a díky nim lze pochopit výkonnostní charakteristiky programu. Tyto nástroje se využijí jak pro měření výkonu, tak při hledání kritických míst v programu při optimalizaci. Významnou částí těchto nástrojů je možnost vizualizace průběhu programu. [16]

### 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. [16]

Graphical user interface, table, Excel

Description automatically generated

Obr. : Grafické prostředí programu Intel VTune. [17]

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 lze zvolit předem nastavené skupiny dat, které chceme sledovat, a také granularitu těchto skupin pro zjištění různých pro nás významných informací. [17]

Graphical user interface, table

Description automatically generated

Obr. : Okno s µpipe zobrazením. [17]

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 lze rozdělit na *front-end*, ve kterém se odehrává načítání a dekódování instrukcí, a *back-end*, který popisuje samotné vykonávání instrukcí za použití funkčních jednotek procesoru. Za předpokladu, že je známo, že front-end v každém cyklu může vygenerovat až 4 operace, lze 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 stanoveno 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 lze identifikovat i konkrétní řádek kódu, který má na svědomí zpomalení. [17]

Calendar

Description automatically generated

Obr. : Pohled *source code analysis*. [17]

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 lze využít dalších typů analýzy. Dále se dají sledovat *memory bound* potíže, způsobené například cache-miss, nebo *core bound*, což popisuje suboptimální využití exekučních jednotek procesoru. [18]

### 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. [16]

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 poskytnou 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 nástroje prostřednictvím síťové komunikace. Aplikaci lze sledovat v reálném čase, také po jejím skončení, nebo může uživatel do aplikace načíst dříve zaznamenané profilovací logy. [19]

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 je třeba zavolat po alokaci paměti a uvolňovací funkci zase před samotným uvolněním paměti. Dále existují funkce pro profilování aplikace využívající technologií 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. [19]

Text

Description automatically generated with medium confidence

Obr. : Použití programu Tracy z příkazové řádky. [Zdroj vlastní]

A picture containing graphical user interface

Description automatically generated

Obr. : 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.

Graphical user interface, text

Description automatically generated

Obr. : Informace o zóně. [Zdroj vlastní]

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

Text

Description automatically generated

Obr. : 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 lze 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.

Graphical user interface, text

Description automatically generated

Obr. : Pohled na program v podobě jazyka symbolických adres. [Zdroj vlastní]

|  |  |
| --- | --- |
|  | Praktická část |

# použité technologie

## Programovací jazyk 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 je třeba s proměnnými pracovat, zpravidla se k tomu využijí *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. [20]

Jazyk C++ byl zvolen k implementaci z důvodu rozšířenosti použití v praxi a možnosti relativně jednoduše uvažovat nad převodem kódu do jazyka symbolických adres. Rovněž tento jazyk můžeme překládat více překladači a díky tomu porovnat vhodnost každého z nich pro různé úlohy.

## 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. [21]

A screenshot of a computer

Description automatically generated with medium confidence

Obr. : 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. Prostředí 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.

Podle ankety uživatelů z roku 2018 bylo toto vývojové prostředí nejpopulárnější mezi vývojáři. Obdrželo vysoké hodnocení zejména v oblasti UX, rozšiřitelnosti a celkovém hodnocení. Na rozdíl od jiných editorů je první setkání s ním hodnoceno kladně, ale za některými zase zaostává po výkonové stránce. [22]

## 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 dovedností 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. [23]

Graphical user interface, application, Word

Description automatically generated

Obr. : Vývojové prostředí Visual Studio 2019. [23]

## 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 rozšířitelný o nové funkce. [24]

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 knihovou. [25]

Text, letter

Description automatically generated

Obr. : Konfigurační soubor kořenového adresáře. [25]

Text, letter

Description automatically generated

Obr. : Konfigurační soubor knihovny Hello. [25]

Text, letter

Description automatically generated

Obr. : Konfigurační soubor adresáře se spustitelným souborem. [25]

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

Tento sestavovací systém byl vybrán kvůli svému rozšíření, podpoře a přenositelnosti.

## 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. Ačkoliv se nejedná o operační systém stejně flexibilní jako Linux, tento operační systém byl zvolen z důvodu svého rozšíření na PC v domácnostech a ve společnostech. Pro představení následujících příkladů, které demonstrují vliv tvorby programů na výkon, je dostačující. Následující údaje byly pořízeny pomocí program CPU-Z.

Graphical user interface, application

Description automatically generated

Obr. : Informace o CPU. [Zdroj vlastní]

Graphical user interface, application

Description automatically generated

Obr. : Informace o operační paměti. [Zdroj vlastní]

Graphical user interface

Description automatically generated

Obr. : Informace o grafické kartě #1. [Zdroj vlastní]



Obr. : Informace o grafické kartě #2. [Zdroj vlastní]

# 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.

## 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 využití instrukce podmíněného skoku. Nejprve se provede vyhodnocení ukončovací podmínky a posléze buď pokračuje do další iterace, nebo smyčka končí.

### Iterace dvourozměrným polem

Velmi jednoduchá ukázka spolupráce s charakteristikami hardware jsou odlišné způsoby iterace přes dvourozměrné pole. Zdrojový kód 1 ukazuje dva základná přístupy. 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.“ [6]

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  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | #define ROWS 5000  #define COLS 5000  double \*\*array;  // Array initialization  double sum = 0.0;  for (int i = 0; i < ROWS; i++)  {  for (int j = 0; j < COLS; j++)  {  sum += array[i][j];  }  }  sum = 0.0;  for (int j = 0; j < COLS; j++)  {  for (int i = 0; i < ROWS; i++)  {  sum += array[i][j];  }  } |

Zdrojový kód : Průchod dvourozměrným polem. [Zdroj vlastní]

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. Zdrojový kód 2 znázorňuje různé přístupové vzorce.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91 | double \*\*A;  double \*\*B;  double \*\*C;  double sum = 0.0;  int n = // A number large enough to ensure one row does not fit the L1 cache  // Version 1  for (int i = 0; i < n; i++)  {  for (int j = 0; j < n; j++)  {  sum = 0.0;  for (int k = 0; k < n; k++)  {  sum += A[i][k] \* B[k][j];  }  C[i][j] = sum;  }  }  // Version 2  for (int j = 0; j < n; j++)  {  for (int i = 0; i < n; i++)  {  sum = 0.0;  for (int k = 0; k < n; k++)  {  sum += A[i][k] \* B[k][j];  }  C[i][j] = sum;  }  }  // Version 3  double r = 0.0;  for (int j = 0; j < n; j++)  {  for (int k = 0; k < n; k++)  {  r = B[k][j];  for (int i = 0; i < n; i++)  {  C[i][j] += A[i][k] \* r;  }  }  }  // Version 4  for (int k = 0; k < n; k++)  {  for (int j = 0; j < n; j++)  {  r = B[k][j];  for (int i = 0; i < n; i++)  {  C[i][j] += A[i][k] \* r;  }  }  }  // Version 5  for (int k = 0; k < n; k++)  {  for (int i = 0; i < n; i++)  {  r = A[i][k];  for (int j = 0; j < n; j++)  {  C[i][j] += r \* B[k][j];  }  }  }  // Version 6  for (int i = 0; i < n; i++)  {  for (int k = 0; k < n; k++)  {  r = A[i][k];  for (int j = 0; j < n; j++)  {  C[i][j] += r \* B[k][j];  }  }  } |

Zdrojový kód : Různé způsoby průchodu maticí při sčítání matic. [6]

### Závislost iterací

Iterace, následující po sobě ve smyčce, mohou mít datové závislosti na výsledcích iterací předchozích, čímž dojde k vynucení sekvenčního provádění a tím pádem nevyužití potenciálu paralelizace smyčky. Jak je zřejmé z představené problematiky v teoretické sekci, v některých případech je možné smyčku přepsat tak, aby mohla být provedena paralelně. Zdrojový kód 3 zobrazuje smyčku se závislostí a bez závislosti.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | double \*A;  double \*B;  double \*C;  double \*D;  int n = // A number large enough to show the impact  // With dependence  for (int i = 0; i < n; i++)  {  A[i] = A[i] + B[i];  B[i+1] = C[i] + D[i];  }    // Without dependence  A[0] = A[0] + B[0];  for (int i = 0; i < n - 1; i++)  {  B[i+1] = C[i] + D[i];  A[i+1] = A[i+1] + B[i+1];  }  B[n] = C[n-1] + D[n-1]; |

Zdrojový kód : Smyčka se závislostí a bez závislosti. [7]

### Loop unrolling

Pro zvýšení počtu instrukcí provádějících pro nás užitečnou práci využijeme několikanásobné duplikování operací v těle smyčky pro několik po sobě následujících iterací. Zdrojový kód 4 demonstruje *loop unrolling 2x1*.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | void loop\_unrolling\_slow(double\* array, int n, double\* result) {  for (int i = 0; i < n; i++)  {  \*result += array[i];  }  }  void loop\_unrolling\_fast(double\* array, int n, double\* result) {  int i = 0;  int limit = n - 1;  double accumulator = 0.0;  // Unrolled loop  for (; i < limit; i+=2)  {  accumulator += array[i] + array[i+1];  }  // Finalizing loop  for (; i < n; i++)  {  accumulator += array[i];  }    \*result = accumulator;  } |

Zdrojový kód : Loop unrolling. [6]

Funkce disponující unrolled smyčkou využívá mimo jiné také optimalizaci v podobě akumulace hodnot do lokální proměnné, čímž potenciálně eliminuje opakované přístupy do paměti z důvodu využití registru procesoru. K zapsání výsledné hodnoty do výsledné proměnné se provede až na samotném konci funkce. Rovněž je zde uveden obecný tvar pro unrolling, což může být zřejmé z druhé, finalizační smyčky. V závislosti na míře unrollingu a hodnotě proměnné *n* nemusí být všechny prvky pole zpracovány v hlavní smyčce. Jak bylo zmíněno v teoretické části, lokálních akumulačních proměnných můžeme deklarovat více, ale je třeba myslet na fyzické možnosti procesoru v podobě omezeného počtu využitelných registrů. Rovněž můžeme zvýšit stupeň unrollingu vícenásobnou duplikací těla smyčky a odpovídajícího navýšení kroku iterační proměnné.

### Projevy asociativity vyrovnávacích paměti

Při konstrukci vyrovnávacích pamětí procesoru je zpravidla využito set-associative vyrovnávacích pamětí s daným stupněm asociativity. Dále bývá nejvyšší úroveň vyrovnávací paměti sdílena mezi všemi jádry. Je třeba myslet na to, že počet míst pro data v jedné sadě je omezený a že existuje možnost thrashingu při opakovaném odkazování na data, která náleží do stejné sady. Zdrojový kód 5 se snaží znázornit tuto skutečnost.

V mnoha případech používají programátoři při tvorbě smyček s nejednotkovým krokem krok, který je mocnina dvou. Při opakované inkrementaci můžeme při převodu výsledné iterační hodnoty do dvojkové soustavy sledovat, že ke změně dochází zejména při vyšších bitech. Ovšem k výběru sady bývají používány bity nižší, a ty se v tomto případě nemění. [26]

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | // Steps: 30, 32, 60, 64, 120, 128, 250, 256, 510, 512  void cache\_associativity\_limit(int step) {  double \*array;  int n = // A number to ensure the same number of memory accesses for every step  // Array initialization  double sum = 0.0;  for (int i = 0; i < n; i += step)  {  sum += array[i];  }  } |

Zdrojový kód : Průchod smyčkou s různým krokem. [Zdroj vlastní]

## Předvídatelnost operací

Moderní hardware využívá prefetchingu pro spekulativní načtení dat do paměti. Dokáže také vypozorovat různé pravidelné vzory přístupu k datům. Ne vždy je ale využíván dostatečně předvídatelný postup. Limitace hardware pro předpovídání přístupu k datům bude představen v následujících příkladech. Zdrojový kód 6 prezentuje různé způsoby přístupu do pole.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | double \*array;  int n = // A number to ensure the same number of memory accesses for every variant  // Array initialization  double sum = 0.0;  // Sequential  for (int i = 0; i < n; i++)  {  sum += array[i];  }  // Strided  for (int i = 0; i < n; i += step)  {  sum += array[i];  }  // Pattern  for (int i = 0; i < n; i++)  {  int index = (2 \* i + 5) % n;  sum += array[index];  }  // Random  srand(seed);  for (int i = 0; i < n; i++)  {  int index = rand() % n;  sum += array[index];  } |

Zdrojový kód : Přístupy do pole s různým krokem. [Zdroj vlastní]

## Hot vs cold data

Následující příklad se týká uspořádání dat v datové struktuře. Je zde využito faktu, že ne všechny proměnné jsou odkazovány stejně často. Některé jsou čteny či zapisovány častěji než jiné. Frekventovaně odkazovaná data jsou nazývána jako *hot*. Naproti tomu málo odkazovaná data jsou označována jako *cold*. [1]

Jestliže se často volá nějakou operaci, která pracuje pouze s částí dat z datové struktury, nevyužívá se celé kapacity cache bloku a pro načtení potřebných dat je třeba se častěji odkazovat do paměti. V případě, že je tato operace volána často, dopad na čas jejího běhu může být značný. Zdrojový kód 7 porovnává klasickou deklaraci a rozdělní dat podle frekvence používání.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39 | // Original  struct Data {  double a, b, c, d, e, f, g, h, result;  };  Data\* data;  int n = // Number of elements in data array  // Array initialization  for (int i = 0; i < n; i++)  {  data[i].result = data[i].a + data[i].b \* data[i].c;  }    // Hot and cold data separated  struct DataHot {  double a, b, c, result;  };  struct DataCold {  double d, e, f, g, h;  };  struct Data {  DataHot\* hot;  DataCold\* cold;  };  Data data;  int n = // Number of elements in data array  // Array initialization  for (int i = 0; i < n; i++)  {  data.hot[i].result = data.hot[i].a + data.hot[i].b \* data.hot[i].c;  } |

Zdrojový kód : Separace často odkazovaných a málo odkazovaných dat. [Zdroj vlastní]

## SoA vs AoS

Typickým příkladem rozdílu mezi OOP a DOP je využití *structure of arrays*. Ve Zdrojový kód 8 se nachází deklaraci datové struktury AoS, která značí *array of structures*. Tímto způsobem je poskládána většina tříd v OOP. Následně je vytvořeno pole těchto datových struktur a je použito podle příkladu ve smyčce. V paměti jsou jednotlivé proměnné ve struktuře poskládány za sebe. Jestliže některé z nich nejsou využity při často volané operaci, mrhá se kapacitou cache bloku a je třeba častěji číst data z paměti. Naproti při DOP se využije *structure of arrays*. V tomto případě se vytvoří jedna proměnná pro data, která ale obsahuje pole pro jednotlivé proměnné. Alokuje se tedy každé z nich. Poté se ve smyčce indexuje až do polí v této struktuře. Je tak lépe využito kapacity cache bloku. Potenciálně by mohlo být AoS využito v kombinaci s *loop unrollingem* či SIMD instrukcemi pro využití prostorové lokality. Zdrojový kód 8 poukazuje na rozlišnosti těchto přístupů.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | // Array of structures  struct AoS {  double a, b, c, d, e, f, g, h, result;  };  AoS\* data;  int n = // Number of elements in data array  // Array initialization  for (int i = 0; i < n; i++)  {  data[i].result = data[i].a + data[i].b \* data[i].c;  }  // Structure of arrays  struct SoA {  double\* a;  double\* b;  double\* c;  double\* d;  double\* e;  double\* f;  double\* g;  double\* h;  double\* results;  };  SoA data;  int n = // Number of elements in each array of data  // Array initialization  for (int i = 0; i < n; i++)  {  data.results[i] = data.a[i] + data.b[i] \* data.c[i];  } |

Zdrojový kód : AoS a SoA přístup. [Zdroj vlastní]

Praktickým příkladem použití je při implementaci hash tabulky. V případě, že je statisticky nejčastěji používaná operace vyhledávání mezi klíči, pak je strukturování hash tabulky jako pár klíč-hodnota nehospodárné, když se bere v potaz utilizaci cache bloku. Dává tedy smysl mít separátní pole pro klíče a hodnoty. Po vyhledání klíče a indexaci do pole hodnot nastane cache miss, ale jelikož se jedná o operaci, která nastává méně často než vyhledávání, je to přijatelné řešení. [3]

## SIMD

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38 | #define N // Number of elements in array  // Scalar  double\* array;  double sum = 0.0;  // Array initialization  for (int i = 0; i < N; i++)  {  sum += array[i];  }  // Vector  double\* array = (double\*)\_aligned\_malloc(sizeof(double) \* N, 16);  // SSE specific step  int step = 128 / (sizeof(double) \* 8);  int sumSize = step;  double\* sum = (double\*)malloc(sizeof(double) \* sumSize);  \_\_m128d sumVector = \_mm\_loadu\_pd(sum);  \_\_m128d\* arrayVector = (\_\_m128d\*)array;  int vecIterations = N / step;  for (int i = 0; i < vecIterations; i++)  {  sumVector = \_mm\_add\_pd(arrayVector[i], sumVector);  }  \_mm\_store\_pd(sum, sumVector);  double sumTotal = 0.0;  for (int i = 0; i < sumSize; i++)  {  sumTotal += sum[i];  } |

Zdrojový kód : Suma hodnot v poli: skalární a SIMD způsob. [Zdroj vlastní]

S využitím SIMD instrukcí lze zpracovávat několik dat najednou. V příkladu je ukázka využití knihovny intrin.h. Zdrojový kód 9 ukazuje využití intrinsických funkcí, které se mapují na SIMD instrukce. Jsou využity SSE instrukce. Jelikož jsou kladeny nároky na zarovnání dat, je třeba využít funkce \_aligned\_malloc. Načtení existující proměnné do vektorové proměnné provedeme pomocí \_mm\_loadu\_pd. Jelikož je zpracováno více dat najednou, počet iterací je nižší. Na závěr je třeba sečíst hodnoty obsažené v proměnné pro akumulaci, jelikož obsahuje dvě skalární hodnoty. Příklad uvažuje, že počet prvků v poli je zpracován kompletně hlavní smyčkou. Reálně by pak byla přidána další smyčku, která by zpracovala zbytek s využitím buď vektorového, či skalárního přístupu.

## Zarovnání dat

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | struct Data {  bool a;  int32\_t b;  bool c;  int64\_t d;  bool e;  };  sizeof(Data) // 32  struct DataOrdered {  int64\_t a;  int32\_t b;  bool c, d, e;  };  sizeof(DataOrdered) // 16 |

Zdrojový kód : Vliv pořadí proměnných v datové struktuře na její velikost. [Zdroj vlastní]

Velikost datového typu proměnné klade určité nároky na umístění v paměti. Tento fakt lze pozorovat při deklaraci datové struktury. Zdrojový kód 10 ukazuje, že dvě datové struktury, které mají identické členské proměnné, ale jejich pořadí se liší, mají různou velikost. Zarovnání dat musí být dodrženo jak pro jednotlivé proměnné, tak pro celou datovou strukturu. To je řešeno vkládáním prázdných proměnných. Důvodem zarovnání je, že instrukce jsou navrženy pro práci nad uniformními daty. Obecné doporučení je řadit členské proměnné podle velikosti jejich datového typu sestupně. [26]

## Datové a kontrolní závislosti

Zdrojový kód 9 obsahuje datovou závislost. Proměnná sum je odkazována v každé iteraci a její aktuální hodnota je závislá také na její hodnotě předchozí. Jedná se tedy o *loop-carried dependence*. Odstranit tento nedostatek lze pomocí rozvinutí této proměnné do vektoru a sumu každé iterace uložit na odpovídající pozici v jednorozměrném poli. Ovšem poté bude třeba tyto mezivýsledky sečíst v další smyčce. [7]

Zdrojový kód 4 představuje kontrolní závislosti, které se dají redukovat pomocí aplikace *loop unrollingu*. Větvení programu může mít mnohdy zásadní vliv na výkon programu a díky snížení množství jejich výskytu se jedná o vhodnou optimalizaci. Hardware se dále snaží snižovat dopad větvení pomocí předvídání, zda se kód větve provede (*taken*), nebo ne (*not taken*). Programátor může pomoci hardware tím, že podmínky ve větvích budou co nejvíce předvídatelné. [7]

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | double\* array = (double \*)malloc(sizeof(double) \* N);  srand(0);  // Initialize  for (int i = 0; i < N; ++i)  {  array[i] = rand() % 10;  }  // Process predictable  for (int i = 0; i < N; ++i)  {  if (array[i] > 20.0) {  printf("%f", array[i]);  }  }  // Process predictable  for (int i = 0; i < N; ++i)  {  if (array[i] >= 0.0 && array[i] < 10.0) {  printf("%f", array[i]);  }  }  // Process not so predictable  for (int i = 0; i < N; ++i)  {  if (array[i] > 5.0) {  printf("%f", array[i]);  }  } |

Zdrojový kód : Předvídatelnost podmínek. [Zdroj vlastní]

Zdrojový kód 11 ukazuje příklady různě předvídatelných podmínek. Alokuje se zde pole s počtem prvků *N* a nastaví se hodnoty pomocí funkce *rand* tak, že žádný prvek nemá hodnotu větší než 10. První zpracovávací smyčka testuje, zda je hodnota prvku větší než 20. Tato podmínka v tomto případě nikdy nenastane. Kompilátor by tuto skutečnost mohl zaznamenat a tuto sekci kódu by vůbec nemusel překládat. Pokud kód odstraněn nebude, alespoň pomůžeme hardware pro spekulaci tím, že větev bude pokaždé *not taken*. Další zpracovávací smyčka má naopak podmínku, která bude pokaždé kladná. V tomto případě bude větev pokaždé *taken*. Vhodná optimalizace by zde byla zcela odstranit instrukce pro kontrolu a využít *loop unrolling*. Poslední zpracovávací smyčka není zdaleka tak předvídatelná, jako ty předchozí. Prostor pro optimalizace je zde výrazně nižší.

## Pravé a falešné sdílení

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41 | #define N // Number of iterations  // True sharing  int shared = 0;  mutex m;  // Do on each core  for (int i = 0; i < N; i++)  {  // Each core increments a shared variable  mutex.lock();  ++shared;  mutex.unlock();  }  // False sharing  int32\_t dataA = 0;  int32\_t dataB = 0;  int32\_t dataC = 0;  int32\_t dataD = 0;  // Do on each core  for (int i = 0; i < N; i++)  {  // Each core increments a not shared variable  ++dataA;  }  // False sharing fix  #define ALIGNMENT (64)  alignas(ALIGNMENT) int32\_t dataAlignedA = 0;  alignas(ALIGNMENT) int32\_t dataAlignedB = 0;  alignas(ALIGNMENT) int32\_t dataAlignedC = 0;  alignas(ALIGNMENT) int32\_t dataAlignedD = 0;  // Do on each core  for (int i = 0; i < N; i++)  {  // Each core increments a not shared variable  ++dataAlignedA;  } |

Zdrojový kód : Pravé a falešné sdílení. [Zdroj vlastní]

Zdrojový kód 12 znázorňuje vliv udržování koherence vyrovnávacích pamětí na výkon aplikace. První příklad je ukázka pravého sdílení dat. Čtyři vlákna procesoru přistupují ke sdílené proměnné. Další ukázka představuje falešné sdílení. Každé vlákno nyní inkrementuje jinou proměnnou, ale jelikož jsou tyto proměnné umístěny v paměti hned vedle sebe, je zde pravděpodobnost, že se vyskytují ve stejném cache bloku. Při přístupu každého vlákna k proměnné dojde k invalidaci bloku a následný cache miss pro ostatní vlákna. Způsob opravy tohoto jevu je umístit každou proměnnou do odlišného cache bloku pomocí zarovnání. [27]

## Aliasing paměti

Při vytváření funkce v jazyce C++ lze předávat jak hodnotu, tak odkaz do paměti. Při využití ukazatelů jako argumentů funkce vzniká omezení pro překladač, který nemůže aplikovat různé optimalizace. Ukazatele by totiž mohly ukazovat na stejné místo v paměti. Toto bývá označováno jako *memory aliasing*. Pokud oba argumenty obsahují stejnou hodnotu, je třeba dodržet určité pořadí operací.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | void write\_read(double\* src, double\* dest, int n) {  int count = n;  int val = 0;  while (count) {  \*dest = val;  val = (\*src) + 1;  count--;  }  }  // Usage  #define N // Number of writes and reads  double \*array;  // Array initialization  // Independent  write\_read(&array[0], &array[1], N);  // Aliased  write\_read(&array[0], &array[0], N); |

Zdrojový kód : Demonstrace aliasingu paměti. [6]

Zdrojový kód 13 ilustruje *write/read dependency*, kde výsledek čtení závisí na nedávné operaci zápisu. Pro pochopení charakteristiky rozdílné doby běhu mezi volání funkce *write­\_read* s argumenty, které ukazují buď na odlišné nebo stejné místo v paměti, je třeba si uvědomit způsob využití funkčních jednotek v CPU. Jednotky pro načítání a čtení dat z paměti mohou pracovat nezávisle na sobě. To platí pouze v případě, že čtení nezávisí na předchozím zápisu. Jednotka pro čtení při své operaci kontroluje *store buffer* jednotky pro zápis pro zjištění, zda není požadavek na zápis na určité místo v paměti.Jestliže se ve *store bufferu* nachází odpovídající záznam, jednotka pro čtení musí vyčkat na provedení zápisu, jinak může pokračovat ve čtení. Jedná se o další ukázku ILP. [6]

## Návrhové vzory z pohledu DOP

Návrhové vzory reprezentují „strategie a řešení aktivit, se kterými se často setkáváme.“ Mohou být použity v různých programovacích jazycích, ale neexistují jako šablonová řešení z důvodu, že „komplexita nepramení z jazyka samotného, ale kvůli faktu, že řešení pochází z problémové domény. Pokud bychom přestali řešený problém modelovat jako objekty a dívali se na něj v podobě technického návrhu, mnoho návrhových vzorů by nebylo zapotřebí.“ [1]

### Existence based processing

Tento způsob práce s daty „zpracovává každý prvek v homogenní sadě dat.“ Existence based processing respektuje způsob funkce hardware a uvědomuje si výhody možné aplikace různých úrovní paralelismu, stejně jako implikace operování nad prvky v poli, které jsou v paměti umístěny za sebou. Zásadní myšlenka je snížit počet všudypřítomných last-minute podmínek, podle kterých se data buď zpracují nebo ne. Toto je docíleno použitím polí s implicitní booleovskou hodnotou. Stejná logika je aplikována pro odstranění rozhodování na základě hodnot enumerací. Rovněž lze tímto způsobem zaregistrovat obsluhu události, kterou je třeba sledovat a zareagovat na ni. [1]

Zdrojový kód 14 ukazuje dvě pole hodnot. Jedno uchovává hodnoty větší nebo rovny nule a druhé hodnoty menší než nula. Implicitní proměnná je v tomto případě „kladná“ a „záporná“ čísla. V některých případech je výhodnější podoba smyčky na řádku číslo 6. Pokud se ale tato logika opakuje, může se vyplatit udržovat si separovaná pole hodnot a následně je pouze zpracovat a neprovádět redundantní kontrolu hodnot. Opět se tedy snižuje počet instrukcí, které jsou potřeba pouze z důvodu režie smyčky, a tedy zvyšuje se množství pro nás přínosné vykonané práce.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | #define N // Number of elements in array  double \*array;  // Array initialization  for (int i = 0; i < N; ++i)  {  if (array[i] >= 0.0)  {  // Positive logic  }  else  {  // Negative logic  }  }  double \*positive;  double \*negative;  int positiveCount;  int negativeCount;  for (int i = 0; i < positiveCount; ++i)  {  // Positive logic on positive[i]  }  for (int i = 0; i < negativeCount; ++i)  {  // Negative logic on negative[i]  } |

Zdrojový kód : Ukázka existence based processing. [Zdroj vlastní]

### Component based objects

Tento způsob pohledu na objekty, se kterými program pracuje, zcela převrací přístup použitý v objektově orientovaném programování. Místo specifikace objektu jako kontejneru, který obsahuje různé, spolu logicky nesouvisející proměnné, lze vnímat objekt jako výsledek souhrnu *komponent*. Komponentu si lze představit jako třídu nebo strukturu obsahující proměnné, které je logické umístit k sobě. Ta potom zpravidla existuje pospolu s ostatními komponentami stejného typu v poli. Na rozdíl od iniciálního návrhu, je zde využito principů lokality ku prospěchu. Původní objekt je nyní popisován pouze spojením odpovídajících částí. Pro práci s ním stačí využít ID pro indexaci nebo vyhledávání v poli komponentů. Tímto vzniká pomocný objekt pro identifikaci zvaný *entita*, který obsahuje pouze unikátní ID. Jelikož komponenty obsahují pouze data, je třeba implementovat způsob práce s těmito daty. K tomu slouží pomocné objekty zvané *managery*, které iterují přes pole komponent a provádějí odpovídající operace. Tento koncept má nejen výkonové implikace, ale také nám umožňuje triviálním způsobem rozšířit funkcionalitu existujícího kódu. [1]

### Adapter

Návrhový vzor „adapter“ slouží k práci s rozhraním, které není možné upravit například z důvodu nepřístupného zdrojového kódu. Pokud si přejeme využít rozhraní jiného, je třeba vytvořit takovýto pomocný adaptér, který vytvoří z dat, se kterými se pracuje, nová data v požadovaném formátu. [28]

V případě DOP pracujeme s daty, která se nachází v poli nebo podobném kontejneru. Data sama o sobě nemají rozhraní, tudíž není co adaptovat. V případě potřeby lze snadno vytvořit transformaci pro práci s potřebnými daty. [1]

### Composite

Návrhový vzor „composite“ vytváří identické rozhraní pro práci s jednotlivými objekty a se skupinami objektů. [28]

Jelikož data se při použití Component based objects nesoustředí do jednoho objektu, nevzniká problém při práci se skupinami objektů. [1]

### Memento

Návrhový vzor „memento“ je zástupcem behaviorálních návrhových vzorů a je využíván k uchovávání stavu systému pro případnou potřebu návratu do tohoto stavu. Rovněž ho můžeme využít pro implementaci chování *undo* a *redo*. [28]

DOP vidí využití pro tento návrhový vzor, specificky při implementaci *level-of-detail* ve videohrách. Při změnách úrovně detailu se vytvoří komprimovaná sada dat, které mohou být v budoucnu využity pro rekonstrukci obrazu s vyššími detaily. [1]

# ověření výkonu pomocí nástrojů

Tabulka : Výsledky implementace zdrojového kódu 1



Tabulka : Výsledky implementace zdrojového kódu 2



# sada doporučení

TODO

Závěr

TODO

Seznam použité literatury

[1] FABIAN, Richard. *Data-oriented design: software engineering for limited resources and short schedules*. Place of publication not identified: Richard Fabian, 2018. ISBN 978-1-916478-70-1.

[2] *Data-Oriented Design (Or Why You Might Be Shooting Yourself in The Foot With OOP) – Games from Within* [online]. 4. prosinec 2009 [vid. 2023-03-10]. Dostupné z: https://gamesfromwithin.com/data-oriented-design

[3] *CppCon 2014: Mike Acton „Data-Oriented Design and C++"* [online]. 2014 [vid. 2023-03-10]. Dostupné z: https://www.youtube.com/watch?v=rX0ItVEVjHc

[4] *Object-oriented programming - Learn web development | MDN* [online]. 24. únor 2023 [vid. 2023-03-10]. Dostupné z: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object-oriented\_programming

[5] MITCHELL, Brad. What Is Functional Programming and Why Use It? *Coding Dojo* [online]. 13. červenec 2022 [vid. 2023-03-10]. Dostupné z: https://www.codingdojo.com/blog/what-is-functional-programming

[6] BRYANT, Randal E. a David R. O’HALLARON. *Computer systems: a programmer’s perspective*. Third edition. Boston: Pearson, 2016. ISBN 978-0-13-409266-9.

[7] HENNESSY, John L., David A. PATTERSON a Andrea C. ARPACI-DUSSEAU. *Computer architecture: a quantitative approach*. 4th ed. Amsterdam ; Boston: Morgan Kaufmann, 2007. ISBN 978-0-12-370490-0.

[8] *Lecture 25: Prefetching - Carnegie Mellon - Computer Architecture 2015 - Onur Mutlu* [online]. 2015 [vid. 2023-03-10]. Dostupné z: https://www.youtube.com/watch?v=ibPL7T9iEwY

[9] *Lecture 26. More Prefetching and Emerging Memory Technologies - CMU - Comp. Arch. 2015 - Onur Mutlu* [online]. 2015 [vid. 2023-03-10]. Dostupné z: https://www.youtube.com/watch?v=TUFins4z6o4

[10] KUSSWURM, Daniel. *Modern x86 assembly language programming: 32-bit, 64-bit, SSE, and AVX*. New York, NY: Apress, 2014. The expert’s voice in programming. ISBN 978-1-4842-0065-0.

[11] STALLMAN, Richard, M. *Using the GNU Compiler Collection: For gcc version 12.2.0* [online]. B.m.: GNU Press. 2022 [vid. 2023-03-10]. Dostupné z: https://gcc.gnu.org/onlinedocs/gcc-12.2.0/gcc.pdf

[12] *Clang Compiler User’s Manual — Clang 17.0.0git documentation* [online]. [vid. 2023-03-10]. Dostupné z: https://clang.llvm.org/docs/UsersManual.html

[13] TYLERMSFT. *MSVC Compiler Options* [online]. 3. srpen 2021 [vid. 2023-03-10]. Dostupné z: https://learn.microsoft.com/en-us/cpp/build/reference/compiler-options

[14] Intel® Intrinsics Guide. *Intel* [online]. [vid. 2023-03-10]. Dostupné z: https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html

[15] *What is Micro-benchmarking?* [online]. [vid. 2023-03-10]. Dostupné z: https://adservio.fr/post/what-is-microbenchmarking

[16] *The Basics of Profiling - Mathieu Ropert - CppCon 2021* [online]. 2022 [vid. 2023-03-10]. Dostupné z: https://www.youtube.com/watch?v=dToaepIXW4s

[17] Intel® VTuneTM Profiler. *Intel* [online]. [vid. 2023-03-10]. Dostupné z: https://www.intel.com/content/www/us/en/develop/documentation/vtune-help/top.html

[18] Intel® VTuneTM Profiler Performance Analysis Cookbook. *Intel* [online]. [vid. 2023-03-10]. Dostupné z: https://www.intel.com/content/www/us/en/develop/documentation/vtune-cookbook/top.html

[19] TAUDUL, Bartosz. *Tracy Profiler: The user manual* [online]. 26. říjen 2022 [vid. 2023-02-06]. Dostupné z: https://github.com/wolfpld/tracy/releases/latest/download/tracy.pdf

[20] STROUSTRUP, Bjarne. *The C++ Programming Language, 4th Edition*. 4th edition. Westport, Conn: Addison-Wesley Professional, 2013. ISBN 978-0-275-96730-7.

[21] *Documentation for Visual Studio Code* [online]. [vid. 2023-03-10]. Dostupné z: https://code.visualstudio.com/docs

[22] *Code Time Data: Ranking the Top 5 Code Editors in 2019 - /src/ blog* [online]. [vid. 2023-03-10]. Dostupné z: https://www.software.com/src/ranking-the-top-5-code-editors-2019

[23] ANANDMEG. *Overview of Visual Studio* [online]. 10. březen 2023 [vid. 2023-03-10]. Dostupné z: https://learn.microsoft.com/en-us/visualstudio/get-started/visual-studio-ide

[24] *Overview | CMake* [online]. [vid. 2023-03-10]. Dostupné z: https://cmake.org/overview/

[25] *Examples | CMake* [online]. [vid. 2023-03-10]. Dostupné z: https://cmake.org/examples/

[26] *CPU Cache Effects - Sergey Slotin - Meeting C++ 2022* [online]. 2022 [vid. 2023-03-10]. Dostupné z: https://www.youtube.com/watch?v=mQWuX\_KgH00

[27] Performance Implications of False Sharing. *CoffeeBeforeArch.github.io* [online]. 28. prosinec 2019 [vid. 2023-03-17]. Dostupné z: https://coffeebeforearch.github.io/2019/12/28/false-sharing-tutorial.html

[28] NESTERUK, Dmitri. *Design patterns in modern C++: reusable approaches for object-oriented software design*. New York, NY: Springer Science+Business Media, 2018. ISBN 978-1-4842-3602-4.

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 Extenstions |
| SSE |  | Streaming SIMD Extensions |
| AVX |  | Advanced Vector Extensions |
| CPI |  | Cycles Per Instruction |
| IPC |  | Instructions Per Cycle |
| GPU |  | Graphics Processing Unit |
| PMU |  | Performance Monitoring Unit |
| PC |  | Personal Computer |

Seznam obrázků

[Obr. 1: Vývoj doby trvání přístupu do procesorové a operační paměti. [7] 17](#_Toc131522178)

[Obr. 2: Příklad rozdělení adresy. [Zdroj vlastní] 17](#_Toc131522179)

[Obr. 3: Příklad uspořádání a typu cache pamětí – snímek programu CPU-Z. [Zdroj vlastní] 19](#_Toc131522180)

[Obr. 4: Sekvenční vykonávání instrukcí. [Zdroj vlastní] 21](#_Toc131522181)

[Obr. 5: Skalární instrukční pipeline. [Zdroj vlastní] 21](#_Toc131522182)

[Obr. 6: Superskalární instrukční pipeline. [Zdroj vlastní] 22](#_Toc131522183)

[Obr. 7: Data hazard. [Zdroj vlastní] 23](#_Toc131522184)

[Obr. 8: Vnitřní architektura x86-32. [10] 27](#_Toc131522185)

[Obr. 9: Příklad programu v jazyce symbolických adres. [10] 29](#_Toc131522186)

[Obr. 10: Příklad běžné kompilace programu pomocí GCC. [Zdroj vlastní] 29](#_Toc131522187)

[Obr. 11: Příklad běžné kompilace programu pomocí Clang. [Zdroj vlastní] 30](#_Toc131522188)

[Obr. 12: Příklad běžné kompilace programu pomocí MSVC. [Zdroj vlastní] 31](#_Toc131522189)

[Obr. 13: Možnosti využití registru v SIMD. [10] 31](#_Toc131522190)

[Obr. 14: Sada registrů AVX. [10] 32](#_Toc131522191)

[Obr. 15: Program sečtení dvou polí a uložení do výsledného pole. [Zdroj vlastní] 33](#_Toc131522192)

[Obr. 16: Kód varianty s přepínači *-O1 -msse*. [Zdroj vlastní] 33](#_Toc131522193)

[Obr. 17: Kód varianty s přepínači *-O2 -msse*. [Zdroj vlastní] 34](#_Toc131522194)

[Obr. 18: Kód varianty s přepínači *-O2 -msse* a lichým počtem prvků v poli. [Zdroj vlastní] 34](#_Toc131522195)

[Obr. 19: Exekuční engine Haswell CPU. [10] 36](#_Toc131522196)

[Obr. 20: Ukázka *loop-carried* závislosti. [7] 39](#_Toc131522197)

[Obr. 21: Přepis smyčky. [7] 39](#_Toc131522198)

[Obr. 22: Příklad atomické výměny hodnot. [7] 41](#_Toc131522199)

[Obr. 23: Spin lock. [7] 42](#_Toc131522200)

[Obr. 24: Graf průběhu. [6] 44](#_Toc131522201)

[Obr. 25: Graf průběhu s promítnutými hodnotami binárního semaforu. [6] 45](#_Toc131522202)

[Obr. 26: Grafické prostředí programu Intel VTune. [17] 47](#_Toc131522203)

[Obr. 27: Okno s µpipe zobrazením. [17] 48](#_Toc131522204)

[Obr. 28: Pohled *source code analysis*. [17] 49](#_Toc131522205)

[Obr. 29: Použití programu Tracy z příkazové řádky. [Zdroj vlastní] 50](#_Toc131522206)

[Obr. 30: Grafické prostředí programu Tracy. [Zdroj vlastní] 51](#_Toc131522207)

[Obr. 31: Informace o zóně. [Zdroj vlastní] 52](#_Toc131522208)

[Obr. 32: Pohled na funkci ze zdrojového souboru. [Zdroj vlastní] 53](#_Toc131522209)

[Obr. 33: Pohled na program v podobě jazyka symbolických adres. [Zdroj vlastní] 54](#_Toc131522210)

[Obr. 34: Grafické prostředí programu Visual Studio Code. [Zdroj vlastní] 57](#_Toc131522211)

[Obr. 35: Vývojové prostředí Visual Studio 2019. [23] 58](#_Toc131522212)

[Obr. 36: Konfigurační soubor kořenového adresáře. [25] 59](#_Toc131522213)

[Obr. 37: Konfigurační soubor knihovny Hello. [25] 59](#_Toc131522214)

[Obr. 38: Konfigurační soubor adresáře se spustitelným souborem. [25] 60](#_Toc131522215)

[Obr. 39: Informace o CPU. [Zdroj vlastní] 60](#_Toc131522216)

[Obr. 40: Informace o operační paměti. [Zdroj vlastní] 61](#_Toc131522217)

[Obr. 41: Informace o grafické kartě #1. [Zdroj vlastní] 61](#_Toc131522218)

[Obr. 42: Informace o grafické kartě #2. [Zdroj vlastní] 62](#_Toc131522219)

TODO

Seznam tabulek

[Tabulka 1: Výsledky implementace zdrojového kódu 1 80](#_Toc131522220)

[Tabulka 2: Výsledky implementace zdrojového kódu 2 81](#_Toc131522221)

TODO

seznam zdrojových kódů

[Zdrojový kód 1: Průchod dvourozměrným polem. [Zdroj vlastní] 64](#_Toc131522326)

[Zdrojový kód 2: Různé způsoby průchodu maticí při sčítání matic. [6] 66](#_Toc131522327)

[Zdrojový kód 3: Smyčka se závislostí a bez závislosti. [7] 66](#_Toc131522328)

[Zdrojový kód 4: Loop unrolling. [6] 67](#_Toc131522329)

[Zdrojový kód 5: Průchod smyčkou s různým krokem. [Zdroj vlastní] 68](#_Toc131522330)

[Zdrojový kód 6: Přístupy do pole s různým krokem. [Zdroj vlastní] 69](#_Toc131522331)

[Zdrojový kód 7: Separace často odkazovaných a málo odkazovaných dat. [Zdroj vlastní] 70](#_Toc131522332)

[Zdrojový kód 8: AoS a SoA přístup. [Zdroj vlastní] 71](#_Toc131522333)

[Zdrojový kód 9: Suma hodnot v poli: skalární a SIMD způsob. [Zdroj vlastní] 72](#_Toc131522334)

[Zdrojový kód 10: Vliv pořadí proměnných v datové struktuře na její velikost. [Zdroj vlastní] 73](#_Toc131522335)

[Zdrojový kód 11: Předvídatelnost podmínek. [Zdroj vlastní] 74](#_Toc131522336)

[Zdrojový kód 12: Pravé a falešné sdílení. [Zdroj vlastní] 75](#_Toc131522337)

[Zdrojový kód 13: Demonstrace aliasingu paměti. [6] 76](#_Toc131522338)

[Zdrojový kód 14: Ukázka existence based processing. [Zdroj vlastní] 78](#_Toc131522339)

Seznam Příloh

Příloha P I: Název přílohy