**Název:** Využití datově orientovaného přístupu ke tvorbě software

**Abstrakt**

Diplomová práce se zabývá tématem datově orientovaného programování. Jejím cílem je poukázat na to, na které aspekty vývoje software by se měli programátoři soustředit pro efektivní využití dostupných zdrojů. V teoretické části jsou popsány techniky návrhu software a také charakteristiky hardware, které zásadním způsobem ovlivňují výkonnost algoritmů. V praktické části jsou srovnány různé způsoby tvorby programů. Každé porovnání je doplněno o výstupy z nástrojů pro výkonnostní testy a pro profilování spotřebovaných zdrojů.

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

**Úvod**

Objektově orientované programování je asi nejrozšířenější paradigma mezi programátory. Mnoho z nich, včetně mě, se s ním setkali na úplném začátku své kariéry. Jedná se o velmi užitečný nástroj. Každý nástroj má ale svůj účel, a není určen k řešení každého problému. Jedním ze zásadních rozdílu mezi OOP a DOP (datově orientované programování) 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. Pokud porozumíme datům, porozumíme problému. Přeci jenom, programy ve své podstatě slouží k transformaci dat z jedné podoby do druhé. Zároveň se toto paradigma soustředí na charakteristiky hardware, na kterém náš software běží a dbá na efektivní využívání zdrojů. Z tohoto důvodu je tento způsob tvorby programů mimo jiné využíván v herním průmyslu. Právě herní vývojáři tvoří naučné podklady o tomto tématu, ve kterých často 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. Osobně jejich nápady považuji za velmi zajímavé, a proto by tato diplomová práce měla být shrnutím nejzásadnějších myšlenek tohoto paradigmata. V teoretické části bude představeno množství praktik, které lze aplikovat na tvorbu programu. V praktické části budou porovnána běžná řešení běžně řešených problémů a také zhodnoceny výstupy výkonnostních testů a profilovacích nástrojů.

**Příklad – dictionary**

Dictionary je datová struktura, která ukládá data jako páry klíčů a hodnot. Klíč slouží k vyhledávání a přístupu k datům, zatímco hodnota udržuje data, které jsou pro nás významné. Velmi častá operace and dictionary je vyhledávání. Pokud se při této operaci pracuje pouze s klíči, jaká je statistická významnost hodnoty? Zajímá nás pouze ta jedna hodnota, kterou chceme najít přes klíč a všechny ostatní hodnoty nás nezajímají. Z tohoto důvodu bychom neměli dictionary implementovat jako páry klíč-hodnota, ale jako dvě pole. Jedno pro klíče a jedno pro hodnoty. Díky tomu dokážeme lépe využít cache paměť stroje. Jelikož budeme pravděpodobně iterovat přes následující klíče, přijde nám vhod fakt, že už budou načtené v cache paměti. Poté při přistoupení k hodnotě přes index nejspíše nastane cache-miss, ale to nevadí, protože statisticky se tato operace děje méně často.

A picture containing text, whiteboard

Description automatically generated

***Poznámky***

The purpose of all programs, and all parts of those programs, is to transform data from one form to another

If you don’t understand the data, you don’t understand the problem

Understand the problem by understanding the data

Different problems require different solutions

If you have different data, you have a different problem

Understand the hardware so we can reason about the cost of solving the problem

Everything is a data problem

Solving problems you probably don’t have creates more problems you definitely do

The more context you have, the better you can make the solution (gather more data)

Focus on solving the most commonly occurring case statistically

Data-oriented is not a new idea, it’s more of a reminder of the first principles

**The three big lies**

1. Software is the platform
2. Code should be designed around the model of the real world
3. Code is more important than data

**Proposal**

1. Hardware is the platform – different hardware requires different solutions
2. Stems from OO approach – world modelling tries to idealize the problem, but you can’t make the problem simpler than it is, therefore design around the data
3. The only purpose of any code is to transform the data – the programmer’s job is not to write code, but to solve problems

There is no ideal abstract solution to a problem

Don’t try to future-proof

Solve for transforming the data you have given the constraints of the platform – and nothing else

Reading from registers is essentially free, reading from L1 cache is ~3 cycles, from L2 ~20 cycles and from RAM ~200 cycles

We should focus on the 90% of code the compiler cannot help us with and help it with the 10% that it can

Don’t waste cache space – separate cold and hot data and pack the data we need for calculation together

Before

Text

Description automatically generated with medium confidence

After – 10x speedup by using memory efficiently (sqrt still expensive instruction)

Graphical user interface

Description automatically generated

Adding bools into structs can push important data out of the cache line

Hoist all loop-invariant reads and branches

Before

for (int i = 0; i < count; ++i) {

if (update) { … }

}

After

if (update) {

for (int i = 0; i < count; ++i) {

…

}

}

The best code is the code that doesn’t need to exist – do-it-once operations, precompile [1]

In essence, data-oriented design is the practice of designing software by developing transforms for well formed data where well formed is guided by the target hardware and the transforms that will operate on it.

If the ultimate result of an application is data, and all input can be represented by data, and it is recognised that all data transforms are not performed in a vacuum, then a software development methodology can be founded on these principles, the principles of understanding the data, and how to transform it given some knowledge of how a machine will do what it needs to do with data of this quantity, frequency, and it’s statistical qualities.

Data-Oriented Design takes it’s cues from the data that is seen or expected. Instead of planning for all eventualities, or planning to make things adaptable, it uses the most probable input to direct the choice of algorithm. Instead of planning to be extendible, it plans to be simple, and get the job done.

Existence-based-processing is when you process every element in a homogeneous set of data. You run the same instructions for every element in that set. There is no definite requirement for the output in this specification, however, usually it is one of three types of operation: filter, mutation, or emission. A mutation is a one to one manipulation of the data, it takes incoming data and some constants that are setup before the transform, and produces one element for each input element. A filter takes incoming data, again with some constants set up before the transform, and produces one element or zero elements for each input element. An emission is a manipulation on the incoming data that can produce multiple output elements.

Domain knowledge is useful because it allows us to lose some otherwise unnecessarily stored data. It is a compiler’s job to analyse the produced output of code (the abstract syntax tree) to then provide itself with data upon which it can infer and use domain knowledge about what operations can be omitted, reordered, or transformed to produce faster or cheaper assembly.

When optimising software, you have to know what is causing the software to run slower than you need it to run. We find in most cases, data movement is what really costs us the most.

When traversing the hierarchy, this dirty bit causes branching based on data that has only just loaded, usually meaning there is no chance to guess the outcome and thus in most cases, causes a pipeline flush and an instruction lookup. For example, in the /emphGCAP 2009 - Pitfalls of Object Oriented Programming presentation by Tony Albrecht in the early slides he declares that checking a dirty flag is less useful than not checking it as if it does fail (the case where the object is not dirty) the calculation that would have taken 12 cycles is dwarfed by the cost of a branch misprediction (23-24 cycles).

***Odkazy***

[1] [CppCon 2014: Mike Acton "Data-Oriented Design and C++" - YouTube](https://www.youtube.com/watch?v=rX0ItVEVjHc)

[2] FABIAN, Richard. Data-Oriented Design: Software engineering for limited resources and short schedules [online]. 2013-09-26. 2013 [cit. 2022-10-14]