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.

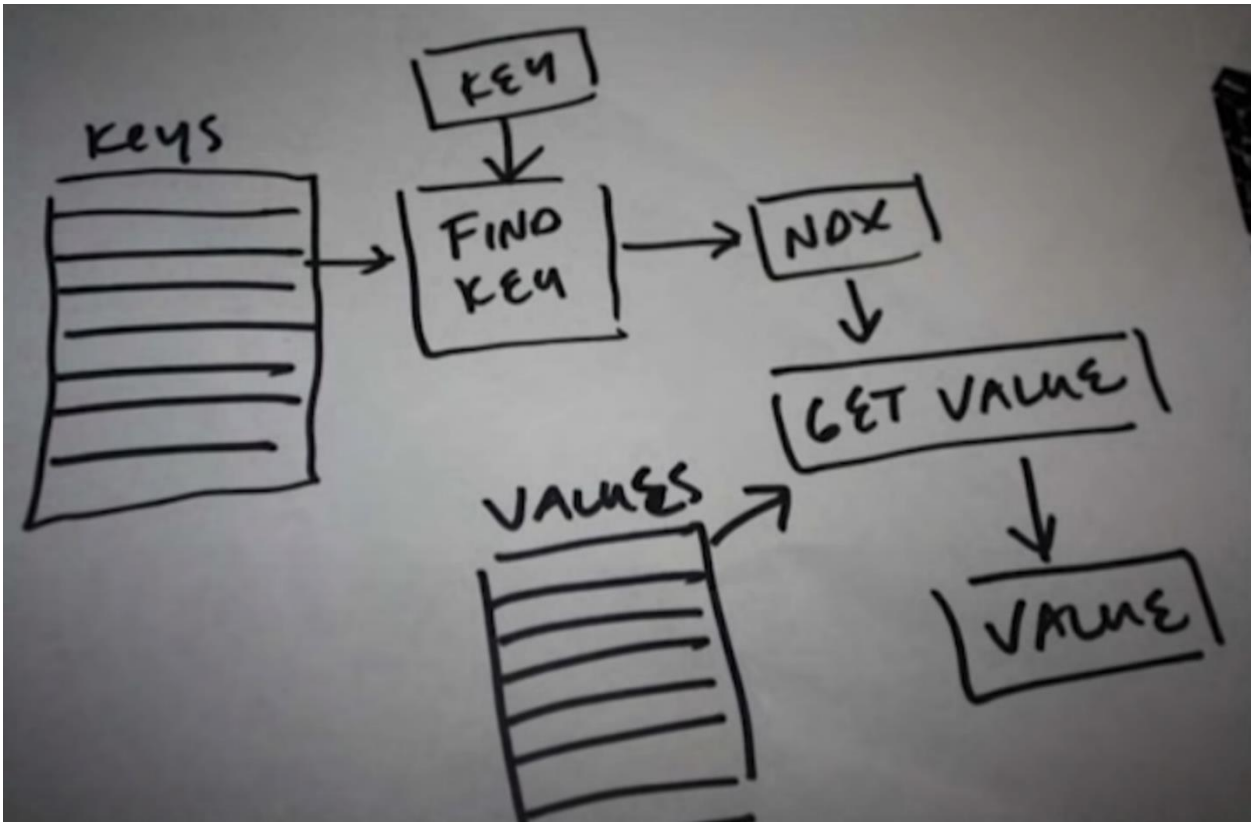
Klíčová slova: data-oriented, C++, optimization, performance, benchmarking, profiling, cache, memory

Úvod

Objektově orientované programování je jedno z nejrozšířenějších paradigmat 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 často 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. [1]



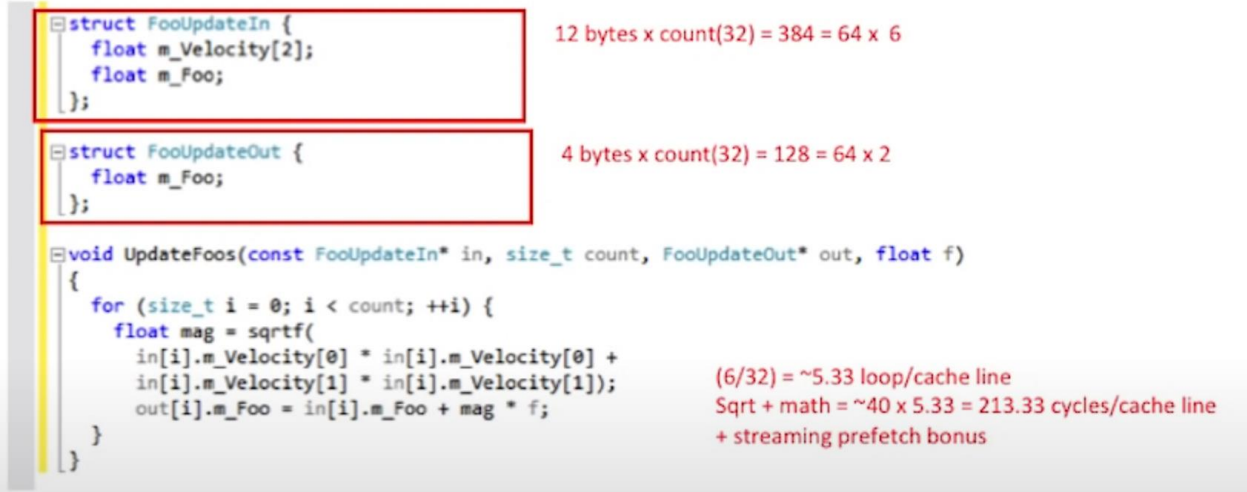
Příklad – separace dat

Before

```
class GameObject {
  float m_Pos[2];
  float m_Velocity[2];
  char m_Name[32];
  Model* m_Model;
  // ... other members ...
  float m_Foo;

  void UpdateFoo(float f)
  {
    float mag = sqrtf(
      m_Velocity[0] * m_Velocity[0] +
      m_Velocity[1] * m_Velocity[1]);
      m_Foo += mag * f;
  }
};
```

After – 10x speedup by using memory efficiently [3]



Příklad – procházení polem

Při procházení dvojrozměrného pole má vliv na výkon fakt, zda iterujeme přes řádky nebo přes sloupce. Při iteraci přes sloupce využíváme skutečnosti, že data jsou v paměti uložena row-major způsobem, tedy data v rámci řádku jsou vedle sebe. Při přístupu k prvnímu prvku pole se načte z paměti jak první prvek, tak i ty následující. Počet následujících načtených prvků je dán velikostí cache-line, která zpravidla bývá 64 B velká. Naopak při iteraci přes řádky existuje šance, že zvláště u větších polí se požadovaný prvek nenachází na stejné cache-line. V takovém případě musí být opět načtena data z paměti, což je relativně dlouhá operace. Procesor v tomto čase musí čekat na její dokončení, zatímco by mohl pokračovat ve výpočtu, kdyby měl data k dispozici.

```
int arr[n][n];

// Rychlý způsob

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

      for (int j = 0; j < n; ++j) {

            array[i][j] += j;

      }

}
```

```
// Pomalý způsob
for (int i = 0; i < n; ++i) {
      for (int j = 0; j < n; ++j) {
            array[j][i] += j;
      }
}
```

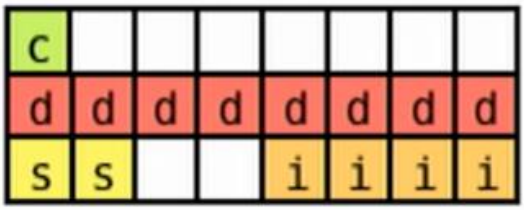
Ještě pomalejší způsob spočívá v náhodném přístupu k řádku. U CPU se vyskytuje prefetcher, který při předvídatelných operacích dokáže přednačíst potřebné data a instrukce. Jelikož ale náhodný přístup není předvídatelný, dojde opět ke zpomalení. [3]

```
// Náhodný přístup
for (int i = 0; i < n; ++i) {
      for (int j = 0; j < n; ++j) {
            array[j][rand()%n] += j;
      }
}
```

Příklad – zarovnání v paměti

Při vytváření vlastních datových struktur např. v jazyce C++ dokážeme ovlivnit její velikost pomocí pořadí deklarovaných členů těchto struktur. Toto je dáno vlastností operačního systému, který jednak udává velikosti primitivních datových typů, ale také udává omezení na to, od které adresy jednotlivé datové typy musejí začínat. Závisí tedy na velikosti „slova“ (word). [3]

```
struct Foo
{
    char c;
    double d;
    short s;
    int i;
};
```




```
struct Foo
{
    char c;
    short s;
    int i;
    double d;
};
```

Příklad – falešné sdílení

Při implementaci vícejádrového algoritmu může dojít k tzv. falešnému sdílení dat. V příkladu jsou deklarovány 4 proměnné, které jsou pak využívány 4 jádry. Problém tohoto příkladu spočívá v tom, že atomické proměnné, jsou v paměti hned za sebou a tím pádem se nacházejí na stejné cache-line. Procesory běžně disponují L3 vyrovnávací pamětí, která je sdílena mezi všemi jádry. Pokud se tedy stejná cache-line nachází v L1 u prvního a druhého jádra, při aktualizaci své atomické proměnné je aktualizována cache-line s ostatními proměnnými a to má za následek zpomalení. Tomuto se dá předejít tak, že každá atomická proměnná se bude nacházet v jiné cache-line. [3]

```
void work (std::atomic<int>& a)
{
    for (int i = 0; i < 10000; ++i)
        a++;
}

void test()
{
    std::atomic<int> a;   a = 0;    // &a: 0x7fff577af220
    std::atomic<int> b;   b = 0;    // &b: 0x7fff577af224
    std::atomic<int> c;   c = 0;    // &c: 0x7fff577af228
    std::atomic<int> d;   d = 0;    // &d: 0x7fff577af22c

    std::thread t1 ([&]() { work (a); });
    std::thread t2 ([&]() { work (b); });
    std::thread t3 ([&]() { work (c); });
    std::thread t4 ([&]() { work (d); });

    t1.join(); t2.join(); t3.join(); t4.join();
}
```

Příklad – pipeline ve smyčce [4]

Some Chain in a Loop

```
for(size_t i=0; i!=n; ++i)
  d(c(b(a(i))));
```

Operation Latencies

- a, b and d: 3
- c: 6

```
for(size_t i=0; i!=n; i += 6) {
  r0 = a(i);
  r1 = a(i+1);
  r2 = a(i+2);
  r3 = a(i+3);  r0 = b(r0);
  r4 = a(i+4);  r1 = b(r1);
  r5 = a(i+5);  r2 = b(r2);
                r3 = b(r3);  r0 = c(r0);
                r4 = b(r4);  r1 = c(r1);
                r5 = b(a4);  r2 = c(r2);
                             r3 = c(r3);
                             r4 = c(r4);
                             r5 = c(r5);
                                          d(r0);
                                          d(r1);
                                          d(r2);
                                          d(r3);
                                          d(r4);
                                          d(r5);
}
```

Příklad – automatická vektorizace smyčky

Pokud ve smyčce neexistuje datová závislost, překladač dokáže sám využít SIMD instrukcí pro zrychlení.

Závislost

```
const int n = 512;
void loop(int* a, int* b, int* c) {
    for (int i=0;i < n; ++i) {
        c[i] = c[i-1] + a[i] + b[i];
    }
}
```

```
loop:
        mov     ecx, DWORD PTR [rdx-4]
        xor     eax, eax
.L2:
        add     ecx, DWORD PTR [rdi+rax]
        add     ecx, DWORD PTR [rsi+rax]
        mov     DWORD PTR [rdx+rax], ecx
        add     rax, 4
        cmp     rax, 2048
        jne     .L2
        ret
n:
        .long   512
```

Bez závislosti

```
const int n = 512;
void loop(int* a, int* b, int* c) {
    for (int i=0;i < n; ++i) {
        c[i] = a[i] + b[i];
    }
}
```

```
loop:
        lea     rcx, [rdi+4]
        mov     rax, rdx
        sub     rax, rcx
        cmp     rax, 8
        jbe     .L5
        lea     rcx, [rsi+4]
        mov     rax, rdx
        sub     rax, rcx
        cmp     rax, 8
        jbe     .L5
        xor     eax, eax
.L3:
        movdqu  xmm0, XMMWORD PTR [rdi+rax]
        movdqu  xmm1, XMMWORD PTR [rsi+rax]
        paddd   xmm0, xmm1
        movups  XMMWORD PTR [rdx+rax], xmm0
        add     rax, 16
        cmp     rax, 2048
        jne     .L3
        ret
.L5:
        xor     eax, eax
.L2:
        mov     ecx, DWORD PTR [rsi+rax]
        add     ecx, DWORD PTR [rdi+rax]
        mov     DWORD PTR [rdx+rax], ecx
        add     rax, 4
        cmp     rax, 2048
        jne     .L2
        ret
n:
        .long   512
```

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

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).

Data-oriented development is not rooted in theory, but practice.

For inputs, make sure you find the optimal way to organise the pre-fetching so that the transform is never starved of data, for outputs, make sure that you write as much as you can in tightly packed consecutive memory to offer opportunity for write combining.

When we think about game entities being objects, we think about them as wholes. But a computer has no concept of objects, and only sees objects as being badly organised data and functions organised for maximal cache abuse.

Object-oriented development is good at providing a human ori ented representation of the problem in the source code, but bad at providing a machine representation of the solution. [2]

Odkazy

[1] CppCon 2014: Mike Acton "Data-Oriented Design and C++" - YouTube

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

[3] https://www.youtube.com/watch?v=BP6NxVxDQIs

[4] An Overview of Program Optimization Techniques - Mathias Gaunard [ACCU 2017] - YouTube