

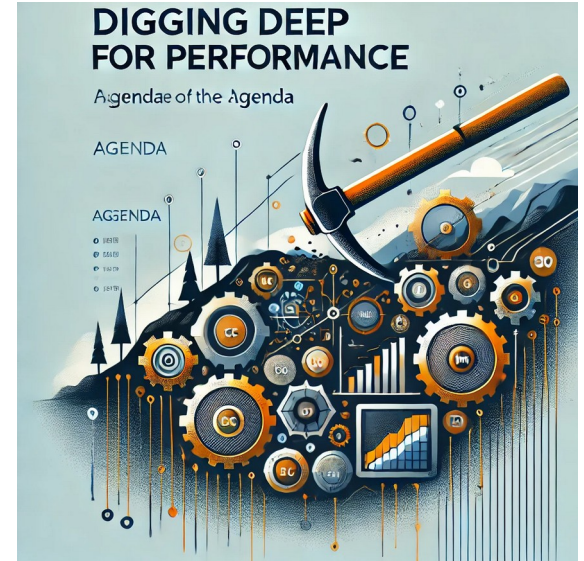


**Q.miners®**

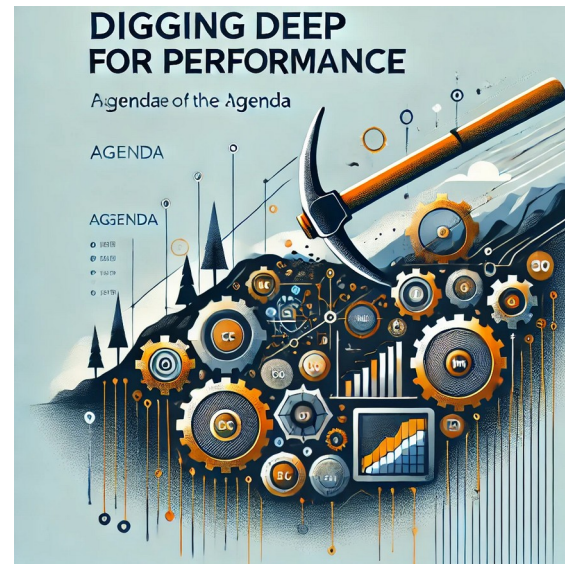
**Digging Deep for Performance**

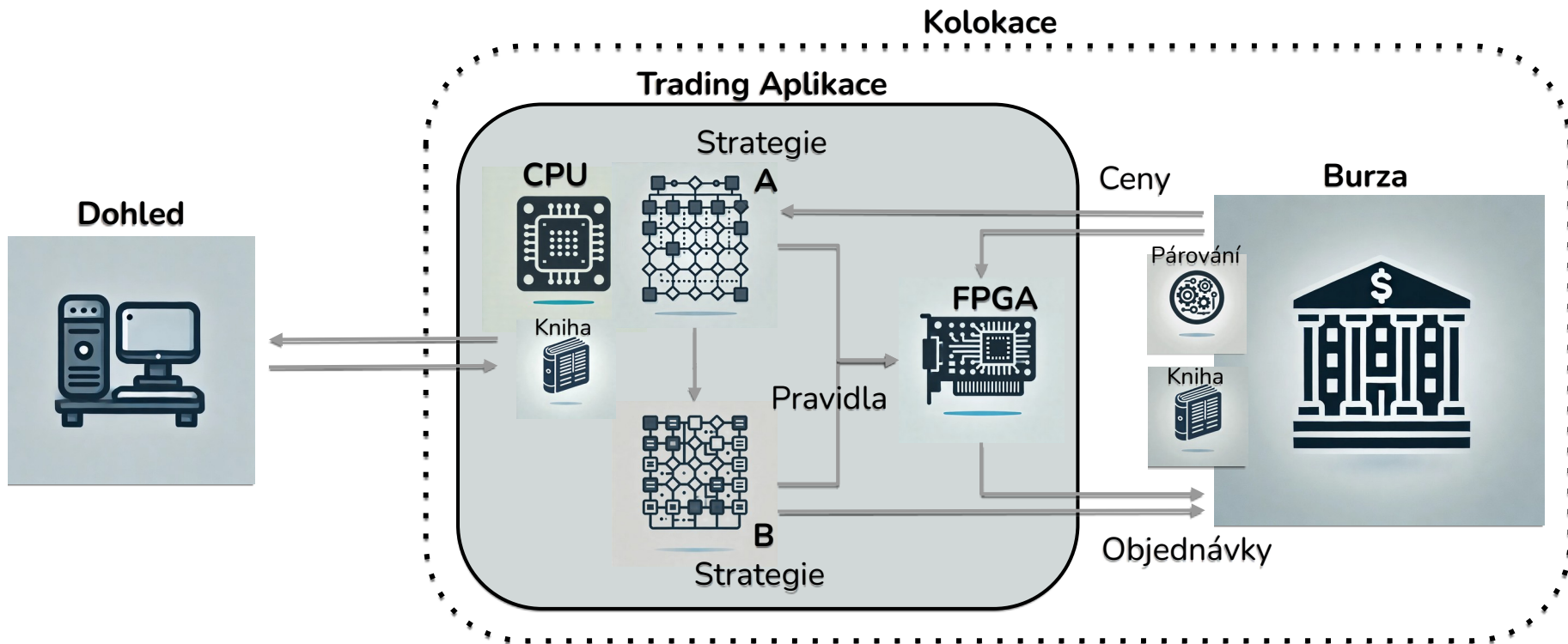
**Hlubkové dolování výkonu**

**Mgr. Petr Filipický**



- Qminers
  - Kdo jsme
  - Co vyvíjíme
  - V čem vyvíjíme
- Case studies (Pár případů z praxe)
  - Trocha teorie
  - Způsoby měření
  - Popis konkrétních situací
- Shrnutí



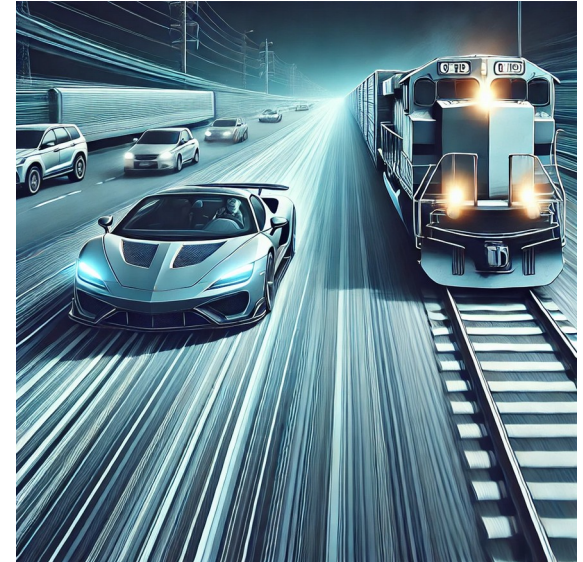


**Latency / Throughput / Bandwidth**



## ● Minimalizace Latence

- Extrémně důležitá v produkci (při samotném obchodování)
- Reakce v řádu desítek mikrosekund  
(existují HW řešení v řádu stovek případně desítek ns)

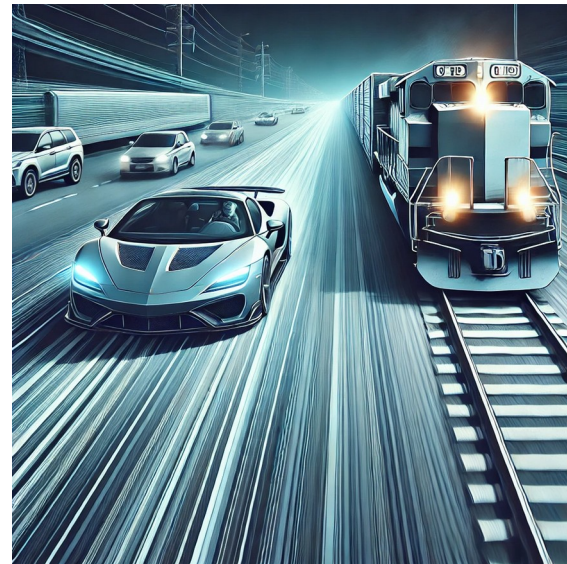
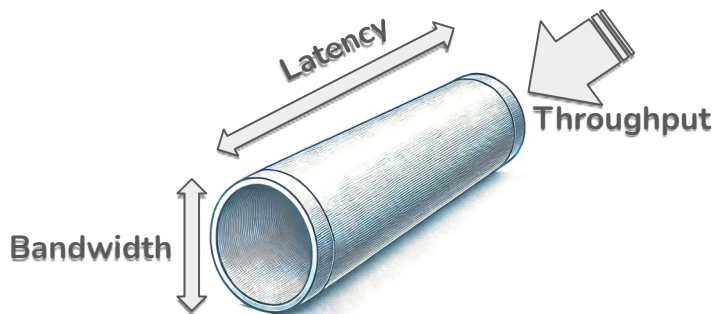


- Minimalizace **Latence**

- Extrémně důležitá v produkci (při samotném obchodování)
- Reakce v řádu desítek mikrosekund  
(existují HW řešení v řádu stovek případně desítek ns)

- Maximalizace **Propustnosti** (Throughput & Bandwidth)

- Důležitá např. při roční simulaci
- Získání výsledků v řádu desítek minut

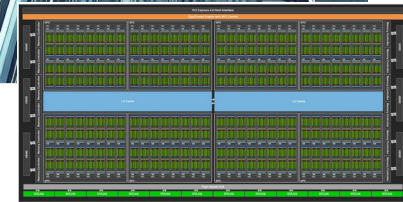
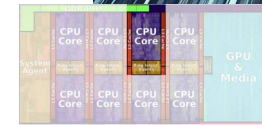
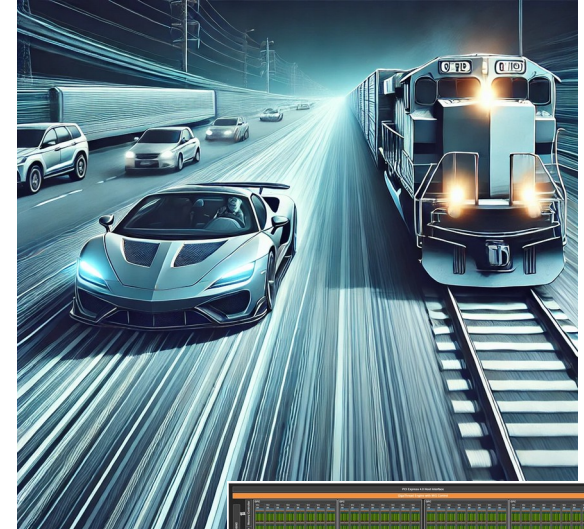
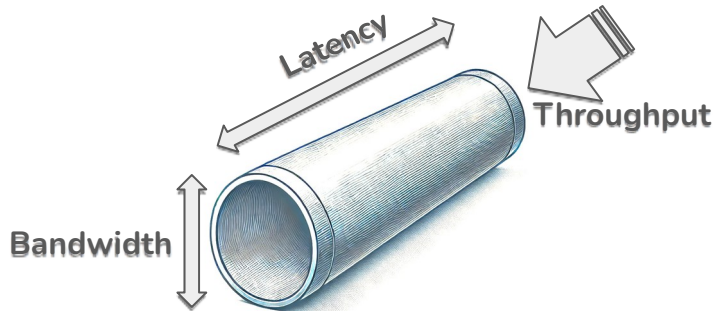


- Minimalizace **Latence**

- Extrémně důležitá v produkci (při samotném obchodování)
- Reakce v řádu desítek mikrosekund  
(existují HW řešení v řádu stovek případně desítek ns)

- Maximalizace **Propustnosti** (Throughput & Bandwidth)

- Důležitá např. při roční simulaci
- Získání výsledků v řádu desítek minut





Call Profile

Search: (No Grouping)

Self	Called	Function
99.70	0.00	(0)
99.70	0.00	43
99.70	0.00	1
99.70	0.00	(0)
99.66	0.00	(0)
99.48	0.00	(0)
98.73	0.00	(0)
98.73	0.00	(0)
98.72	0.24	(0)
97.67	0.67	(0)
75.80	0.68	(0)
75.71	0.17	(0)
74.56	0.10	(0)
73.31	0.03	(0)
73.28	0.66	(0)
72.58	0.03	(0)
72.55	2.65	(0)
72.11	1.00	(0)
56.87	0.52	(0)
56.35	0.02	(0)
56.32	1.98	(0)
37.09	0.01	(0)
37.08	0.58	(0)
30.25	0.18	(0)
30.07	0.95	(0)
27.89	3.98	(0)
27.65	0.08	(0)
27.46	0.13	(0)
27.32	0.46	(0)
18.21	0.16	(0)
18.05	2.78	(0)
15.70	0.38	(0)
15.32	0.36	(0)
15.27	15.17	(0)
12.81	0.01	(0)
12.80	0.12	(0)
12.55	0.15	(0)
12.16	0.35	(0)
11.14	0.29	(0)
10.32	0.00	(0)
10.32	0.15	(0)
10.20	0.19	(0)
9.80	0.48	(0)
8.37	0.17	(0)
8.24	0.46	(0)
7.73	0.17	(0)
7.58	2.68	(0)
7.55	7.55	(0)
7.39	0.02	(0)
7.36	0.03	(0)
6.24	0.01	(0)
6.23	3.07	(0)
5.68	0.17	(0)
5.46	0.18	(0)
5.39	0.11	(0)
5.28	1.22	(0)
3.92	0.04	(0)
3.76	0.01	(0)
3.65	0.22	(0)
3.61	0.58	(0)
3.24	0.18	(0)
3.03	0.02	(0)
3.02	0.15	(0)
2.71	2.71	(0)
2.66	0.01	(0)
2.65	0.58	(0)
2.59	0.16	(0)
2.49	0.00	(0)
2.49	0.00	(0)
2.49	0.00	(0)
2.49	0.02	(0)

# Performance measurement



Qminers:trader:KoinersTrader:run()

Types Callers All Callers Callee Map Source Code

Call Graph

Callers Call Graph All Callers Caller Map Machine Code

Search: (No Grouping)

Self	Called	Function
99.70	0.00	(0)
99.70	0.00	43
99.70	0.00	1
99.70	0.00	(0)
99.66	0.00	(0)
99.48	0.00	(0)
98.73	0.00	(0)
98.73	0.00	(0)
98.72	0.24	1
97.67	0.67	23 2271 225
75.80	0.68	13 160 040
75.71	0.17	13 160 040
74.56	0.10	13 160 040
73.31	0.03	93 897 122
73.28	0.66	93 897 122
72.58	0.03	93 897 122
72.55	2.65	93 897 122
72.11	1.00	13 160 040
56.87	0.52	79 283 520
56.35	0.02	79 283 520
56.32	1.98	79 283 520
37.09	0.01	40 216 638
37.08	0.58	40 216 638
30.25	0.18	27 838 596
30.07	0.95	27 838 596
27.89	0.66	370 546
27.65	0.08	63 770 162
27.46	0.13	40 216 638
27.32	0.46	40 216 638
18.21	0.16	40 216 638
18.05	2.78	40 216 638
15.32	0.36	46 795 698
15.27	15.17	4 638 935
12.81	0.01	47 101 136
12.80	0.12	47 101 136
12.55	0.15	9 586 985
12.16	0.35	16 834 516
11.14	0.29	39 944 645
10.32	0.00	13 158 868
10.32	0.15	13 158 868
10.20	0.19	9 502 887
9.80	0.48	40 369 357
8.37	0.17	39 639 207
8.24	0.46	39 944 645
7.73	0.17	39 639 207
7.58	2.68	39 944 645
7.55	7.55	39 639 207
7.39	0.02	6 579 348
7.36	0.03	6 579 348
6.24	0.01	40 216 638
6.23	3.07	40 216 638
5.68	0.17	19 724 048
5.46	0.18	19 724 048
5.39	0.11	9 502 887
5.28	1.22	18 222 314
3.92	0.04	2 691 233
3.76	0.01	2 608 383
3.65	0.22	9 527 826
3.61	0.58	52 352 847
3.24	0.18	19 840 546
3.03	3.02	52 352 847
3.02	0.15	19 840 546
2.71	2.71	134 899 236
2.66	0.01	46 796 897
2.65	0.58	46 796 897
2.59	0.16	26 320 081
2.49	0.00	53 130
2.49	0.00	53 129
2.49	0.00	53 129
2.49	0.00	1 107
2.49	0.02	36 460

# Performance measurement



Samplování (Linux perf, Intel VTune, ...)



The screenshot displays a performance analysis tool interface. At the top, there are tabs for 'Types', 'Callers', 'All Callers', 'Callee Map', and 'Source Code'. The main area is divided into two sections. The upper section shows a call graph with nodes representing functions and their relationships. The lower section shows a call stack with a tree structure of function calls. The bottom of the window has tabs for 'Callers', 'Call Graph', 'All Callers', 'Callee Map', and 'Machine Code'. The bottom status bar shows 'Total Instruction Fetch Cost: 640 102 367 978'.

Search: (No Grouping)

Self	Called	Function
99.70	0.00	(0)
99.70	0.00	43
99.70	0.00	1
99.70	0.00	0
99.66	0.00	0
99.48	0.00	0
98.73	0.00	0
98.73	0.00	0
98.72	0.14	1
97.67	0.67	23 271 225
75.80	0.68	13 160 040
75.71	0.17	13 160 040
74.56	0.10	13 160 040
73.31	0.03	93 897 122
73.28	0.66	93 897 122
72.58	0.03	93 897 122
72.55	2.65	93 897 122
72.11	1.00	13 160 040
56.87	0.52	79 283 520
56.35	0.02	79 283 520
56.32	1.98	79 283 520
37.09	0.01	40 216 638
37.08	0.58	40 216 638
30.25	0.18	27 838 596
30.07	0.95	27 838 596
27.89	3.98	66 376 546
27.65	0.08	63 770 162
27.46	0.13	40 216 638
27.32	0.46	40 216 638
18.21	0.16	40 216 638
18.05	2.78	40 216 638
17.51	0.15	23 606 496
17.36	0.73	23 606 496
16.16	1.46	93 897 122
15.80	0.04	6 092 428
15.70	0.38	6 092 428
15.32	0.36	46 795 698
15.27	15.17	4 639 935
12.81	0.01	47 101 136
12.80	0.12	47 101 136
12.55	0.15	9 586 985
12.16	0.35	16 834 516
11.14	0.29	39 944 645
10.32	0.00	13 158 868
10.32	0.15	13 158 868
10.20	0.19	9 502 887
9.80	0.48	40 369 357
8.37	0.17	39 639 207
8.24	0.46	39 944 645
7.73	0.17	39 639 207
7.58	2.68	39 944 645
7.55	7.55	39 639 207
7.39	0.02	6 579 348
7.36	0.03	6 579 348
6.24	0.01	40 216 638
6.23	3.07	40 216 638
5.68	0.17	19 724 048
5.46	0.18	19 724 048
5.39	0.11	9 502 887
5.28	1.22	18 222 314
3.92	0.04	2 691 233
3.76	0.01	2 608 383
3.65	0.22	9 527 826
3.61	0.58	52 352 847
3.24	0.18	19 840 546
3.03	3.02	52 352 847
3.02	0.15	19 840 546
2.71	2.71	134 899 236
2.66	0.01	46 796 897
2.65	0.58	46 796 897
2.59	0.16	26 320 081
2.49	0.00	53 130
2.49	0.00	53 129
2.49	0.00	1 107
2.49	0.02	36 460

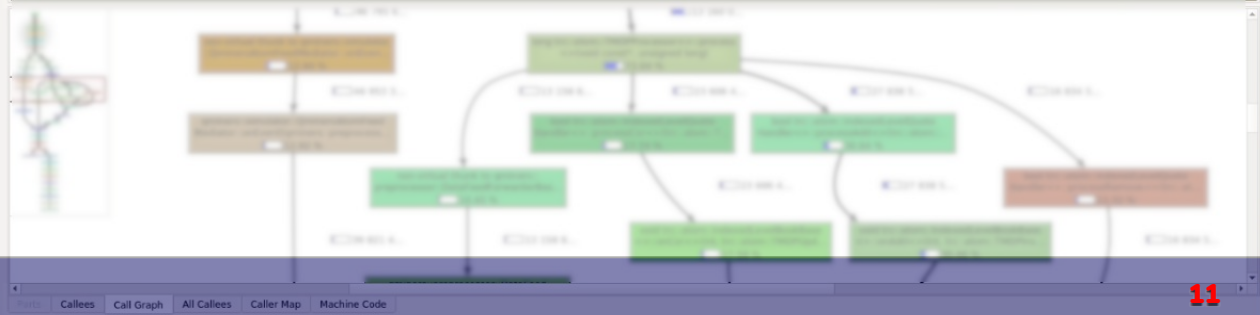
# Performance measurement



Samplování (Linux perf, Intel VTune, ...)



Instrumentace (Optick, ...)



Self	Called	Function
99.70	0.00	(0)
99.70	0.00	43
99.70	0.00	1
99.70	0.00	60
99.66	0.00	0
99.48	0.00	0
98.73	0.00	0
98.73	0.00	0
98.72	0.14	1
97.67	0.67	23 2271 225
75.80	0.68	13 160 040
75.71	0.17	13 160 040
74.56	0.10	13 160 040
73.31	0.03	93 897 122
73.28	0.66	93 897 122
72.58	0.03	93 897 122
72.55	2.65	93 897 122
72.11	1.00	13 160 040
56.87	0.52	79 283 520
56.35	0.02	79 283 520
56.32	1.98	79 283 520
37.09	0.01	40 216 638
37.08	0.58	40 216 638
30.25	0.18	27 838 596
30.07	0.95	27 838 596
27.89	3.98	66 376 546
27.65	0.68	63 770 162
27.46	0.13	40 216 638
27.32	0.46	40 216 638
18.21	0.16	40 216 638
18.05	2.78	40 216 638
17.51	0.15	23 606 496
17.36	0.73	23 606 496
16.16	1.46	93 897 122
15.80	0.04	6 092 428
15.70	0.38	6 092 428
15.32	0.36	46 796 897
15.27	15.17	4 638 935
12.81	0.01	47 101 136
12.80	0.12	47 101 136
12.55	0.15	9 586 985
12.16	0.35	16 834 516
11.14	0.29	39 948 829
10.32	0.00	13 158 868
10.32	0.15	13 158 868
10.20	0.19	9 502 887
9.80	0.48	40 369 357
8.37	0.17	39 639 207
8.24	0.46	39 948 829
7.73	0.17	39 639 207
7.58	2.68	39 944 645
7.55	7.55	39 639 207
7.39	0.02	6 579 348
7.36	0.03	6 579 348
6.24	0.01	40 216 638
6.23	3.07	40 216 638
5.68	0.17	19 724 048
5.46	0.18	19 724 048
5.39	0.11	9 502 887
5.28	1.22	18 222 114
3.92	0.04	2 691 233
3.76	0.01	2 608 383
3.65	0.22	9 527 826
3.61	0.58	52 352 847
3.24	0.18	19 840 546
3.03	3.02	52 352 847
3.02	0.15	19 840 546
2.71	2.71	134 899 236
2.66	0.01	46 796 897
2.65	0.58	46 796 897
2.59	0.16	26 320 081
2.49	0.00	53 130
2.49	0.00	53 129
2.49	0.00	1 107
2.49	0.02	36 460

# Performance measurement



Samplování (Linux perf, Intel VTune, ...)



Instrumentace (Optick, ...)



Emulace (Valgrind)

The screenshot shows the Qminers interface with a call graph on the left and machine code on the right. The call graph displays a hierarchy of function calls, with 'main' at the top, branching into 'main', 'main', and 'main'. The machine code view shows assembly instructions with their corresponding addresses.

Self	Called	Function
99.70	0.00	(0)
99.70	0.00	43
99.70	0.00	1
99.70	0.00	(0)
99.66	0.00	(0)
99.48	0.00	(0)
98.73	0.00	(0)
98.72	0.14	1
97.67	0.67	23 2271 225
75.80	0.68	13 160 040
75.71	0.17	13 160 040
74.56	0.10	13 160 040
73.31	0.03	93 897 122
73.28	0.66	93 897 122
72.58	0.03	93 897 122
72.55	2.65	93 897 122
72.11	1.00	13 160 040
56.87	0.52	79 283 520
56.35	0.02	79 283 520
37.09	1.98	79 283 520
37.09	0.01	40 216 638
37.08	0.58	40 216 638
30.25	0.18	27 838 596
30.07	0.95	27 838 596
27.89	3.98	66 376 546
27.65	0.08	63 770 162
27.46	0.13	40 216 638
27.32	0.46	40 216 638
18.21	0.16	40 216 638
18.05	2.78	40 216 638
17.51	0.15	23 606 496
17.36	0.73	23 606 496
16.16	1.46	93 897 122
15.80	0.04	6 092 428
15.70	0.38	6 092 428
15.32	0.36	46 796 698
15.27	15.17	4 638 935
12.81	0.01	47 101 136
12.80	0.12	47 101 136
12.55	0.15	9 586 985
12.16	0.35	16 834 516
11.14	0.29	39 948 829
10.32	0.00	13 158 868
10.32	0.15	13 158 868
10.20	0.19	9 502 887
9.80	0.48	40 369 357
8.37	0.17	39 639 207
8.24	0.46	39 948 829
7.73	0.17	39 639 207
7.58	2.68	39 944 645
7.55	7.55	39 639 207
7.39	0.02	6 579 348
7.36	0.03	6 579 348
6.24	0.01	40 216 638
6.23	3.07	40 216 638
5.68	0.17	19 724 048
5.46	0.18	19 724 048
5.39	0.11	9 502 887
5.28	1.22	18 222 314
3.92	0.04	2 691 233
3.76	0.01	2 608 383
3.65	0.22	9 527 826
3.61	0.58	52 352 847
3.24	0.18	19 840 546
3.03	3.02	52 352 847
2.02	0.15	19 840 546
2.01	2.71	134 899 236
2.66	0.01	46 796 897
2.05	0.58	46 796 897
2.59	0.16	26 320 082
2.49	0.00	53 130
2.49	0.00	53 129
2.49	0.00	53 129
2.49	0.00	1 107
2.49	0.02	36 460

# Performance measurement



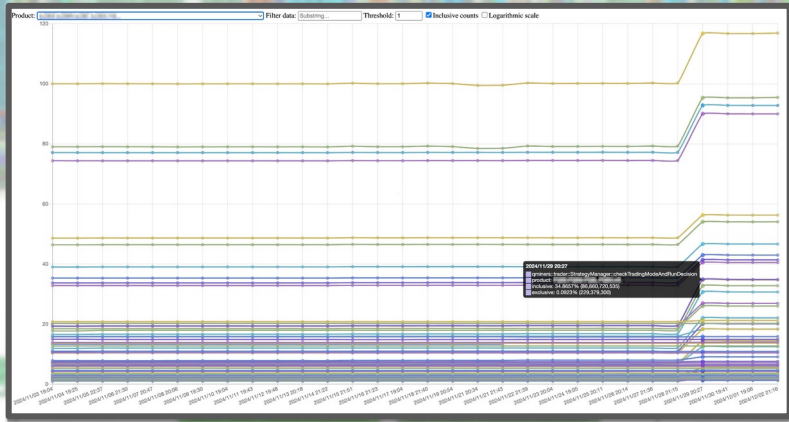
Samplování (Linux perf, Intel VTune, ...)



Instrumentace (Optick, ...)



Emulace (Valgrind)

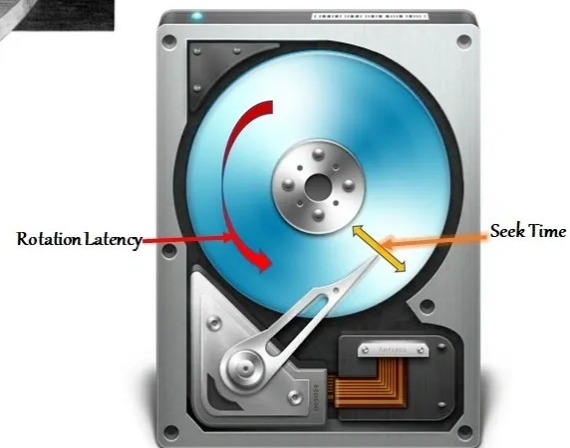


**Memory**

**Q.miners<sup>®</sup>**

- Paměti s lineárním přístupem

- Magnetické pásky
- Plotnové disky

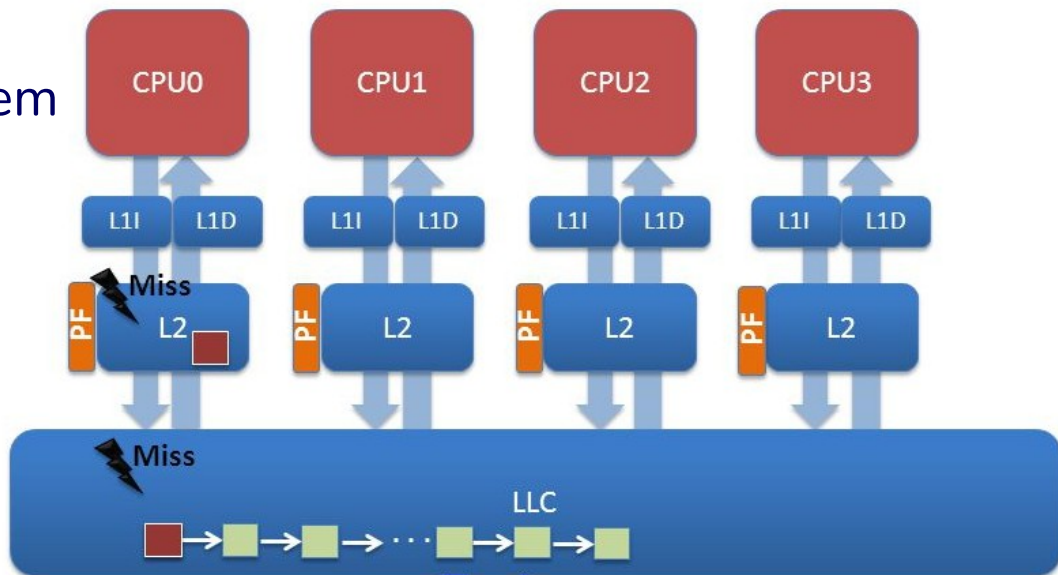


- Paměti s lineárním přístupem

- Magnetické pásky
- Plotnové disky

- Je RAM opravdu RAM?

- Hierarchie vyrovnávacích pamětí
- Různé latence



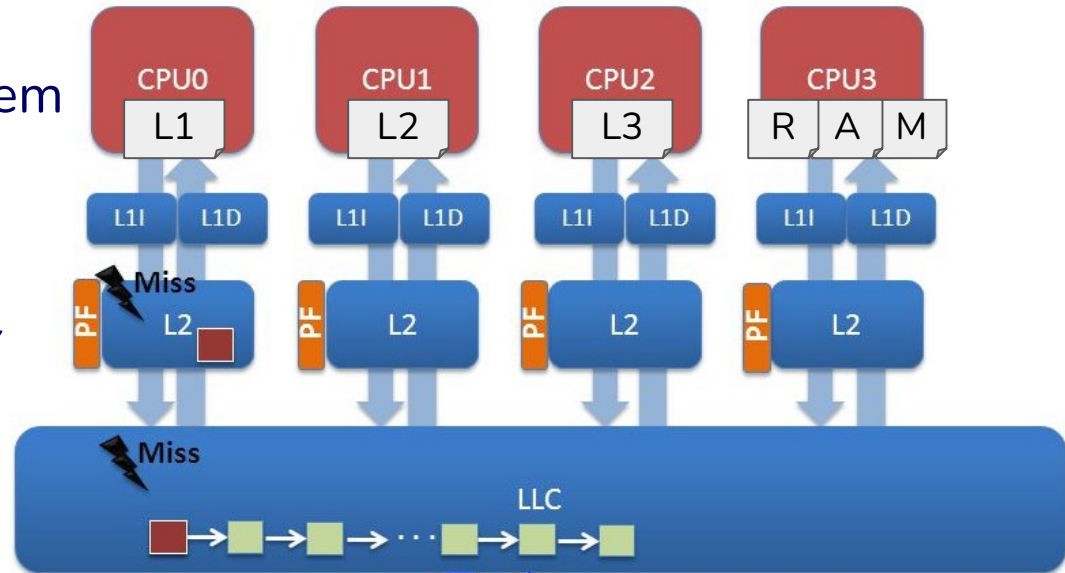


- Paměti s lineárním přístupem

- Magnetické pásy
- Plotnové disky

- Je RAM opravdu RAM?

- Hierarchie vyrovnávacích pamětí
- Různé latence



- Paměti s lineárním přístupem

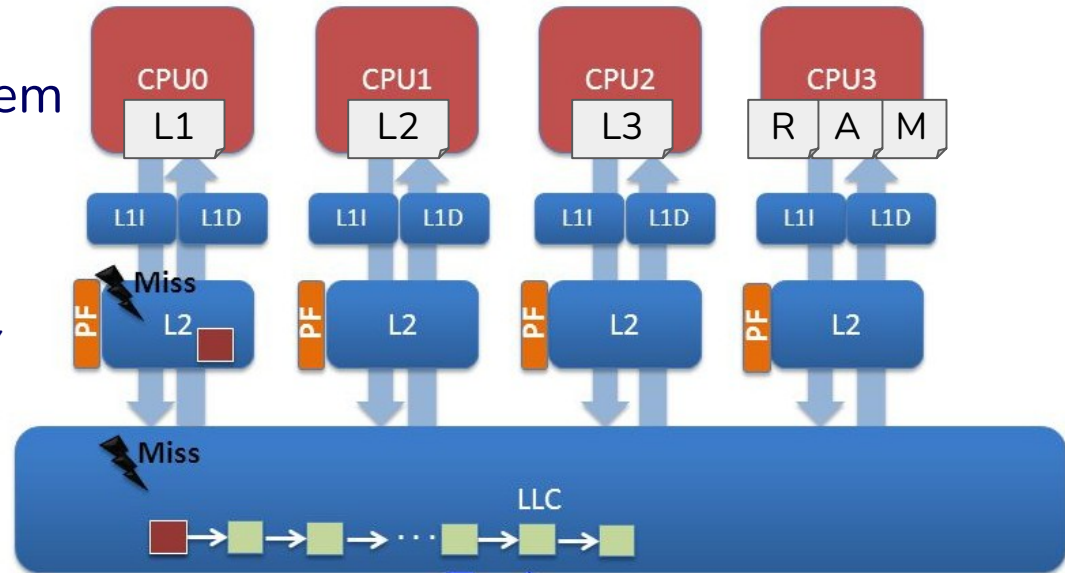
- Magnetické pásky
- Plotnové disky

- Je RAM opravdu RAM?

- Hierarchie vyrovnávacích pamětí
- Různé latence

- Lokalita přístupu

- Časová (Cache)
- Prostorová (Cache line a Prefetcher)



- Paměti s lineárním přístupem

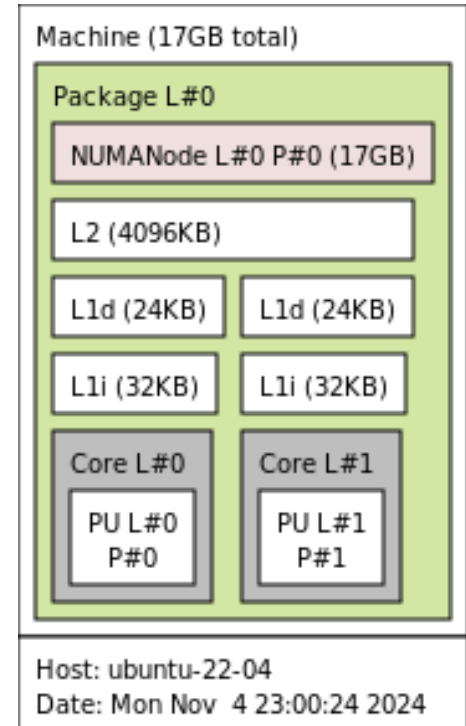
- Magnetické pásky
- Plotnové disky

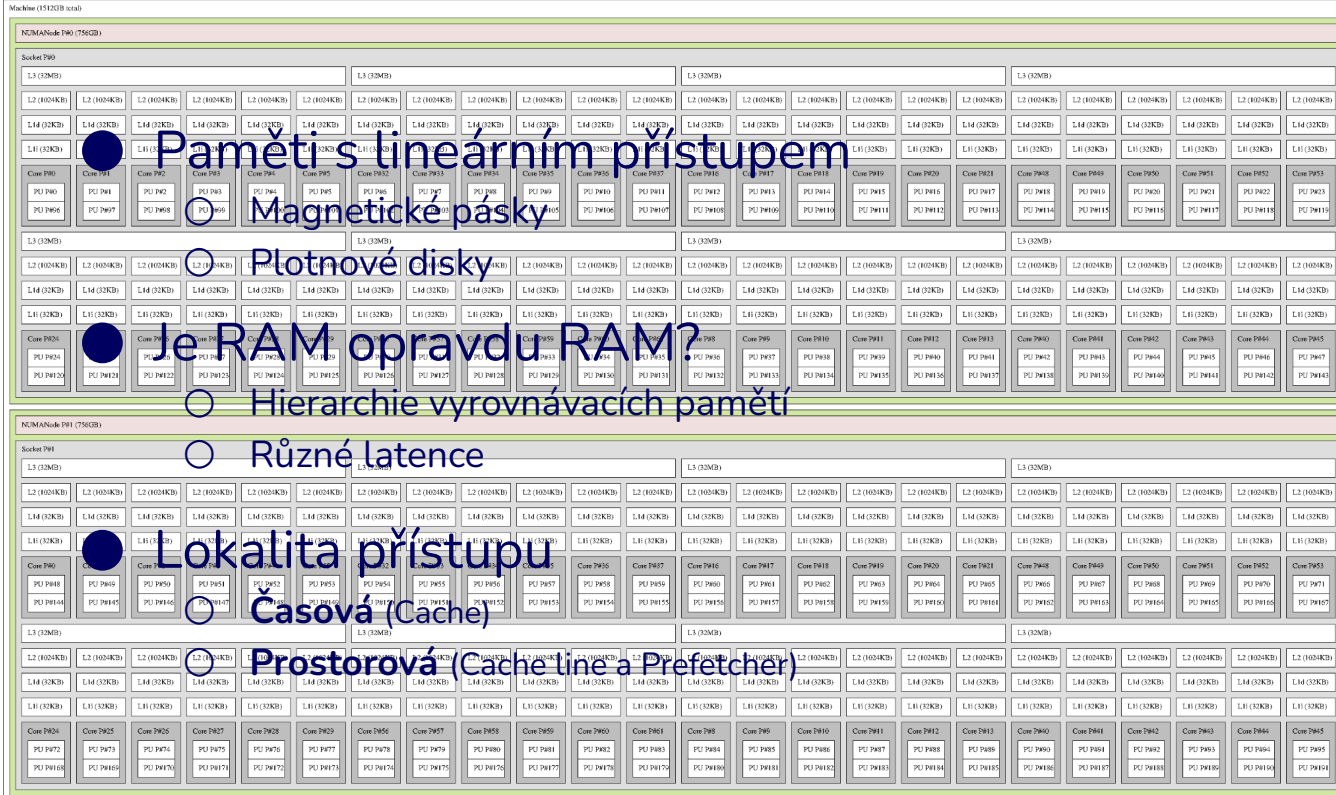
- Je RAM opravdu RAM?

- Hierarchie vyrovnávacích pamětí
- Různé latence

- Lokalita přístupu

- **Časová** (Cache)
- **Prostorová** (Cache line a Prefetcher)





● Paměť s lineárním přístupem

○ Magnetické pásy

○ Plotnové disky

● Je RAM opravdu RAM?

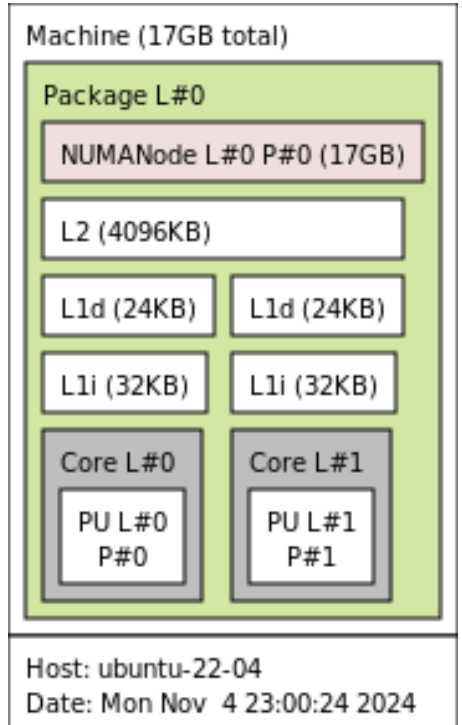
○ Hierarchie vyrovnávacích pamětí

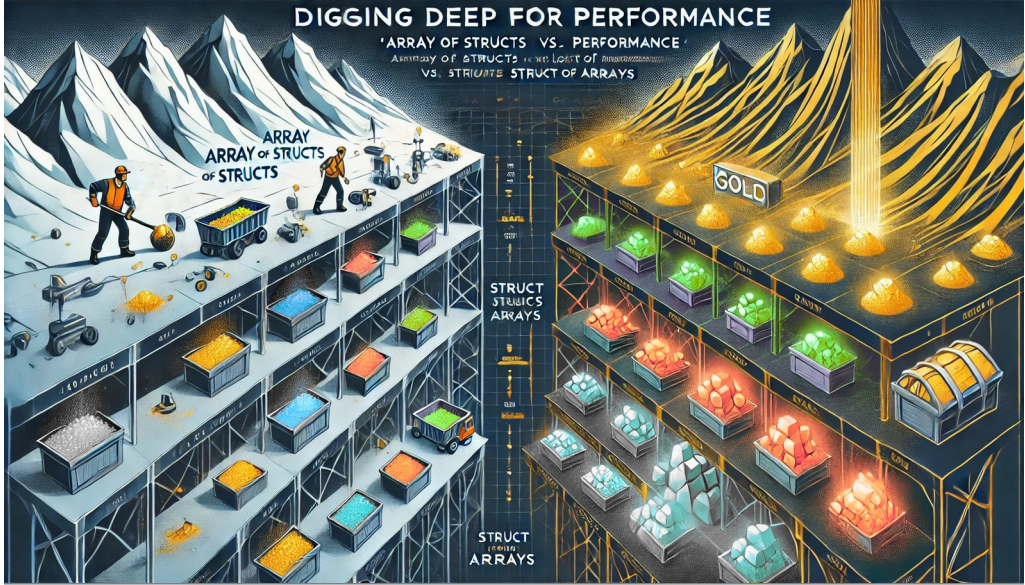
○ Různé latence

● Lokality přístupu

○ Casová (Cache)

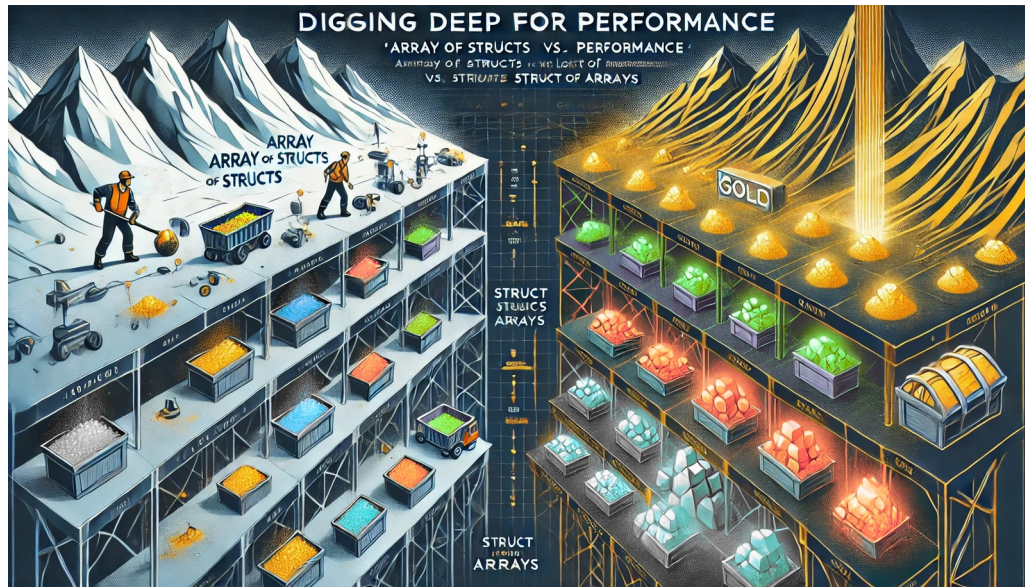
○ Prostorová (Cache line a Prefetcher)





## ● Array of Structs

```
namespace AOS {  
struct Beast {  
    Point3D pos{};  
    Point3D vel{};  
    HealthPoints hp{};  
};  
using Beasts = std::vector<Beast>;
```

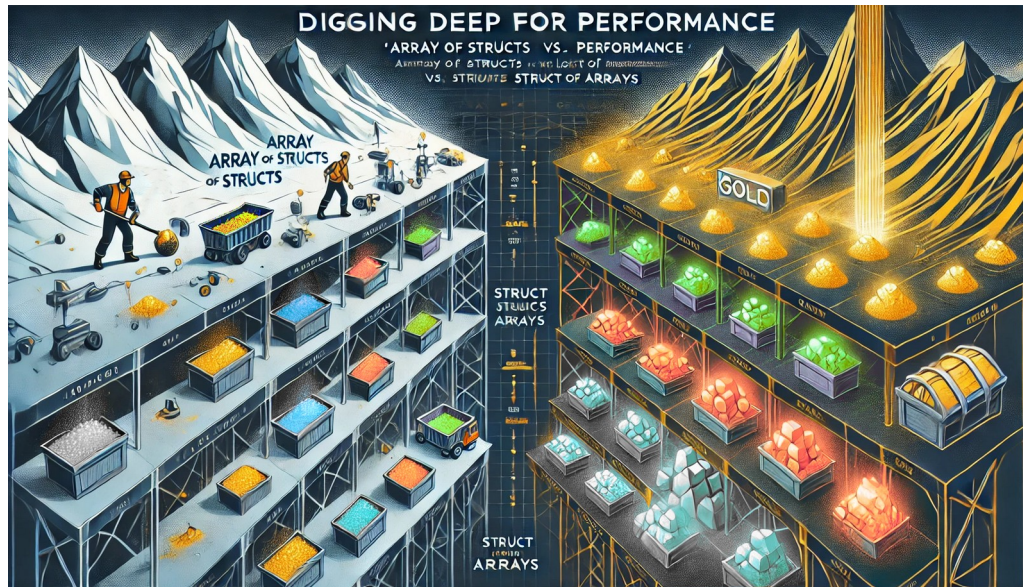


## ● Array of Structs

```
namespace AOS {  
struct Beast {  
    Point3D pos{};  
    Point3D vel{};  
    HealthPoints hp{};  
};  
using Beasts = std::vector<Beast>;
```

## ● Struct of Arrays

```
namespace SOA {  
struct Beasts {  
    explicit Beasts(size_t size)  
        : pos(size), vel(size), hp(size) {}  
    std::vector<Point3D> pos;  
    std::vector<Point3D> vel;  
    std::vector<HealthPoints> hp;  
};
```



## ● Array of Structs

```
namespace AOS {  
struct Beast {  
    Point3D pos{};  
    Point3D vel{};  
    HealthPoints hp{};  
};  
using Beasts = std::vector<Beast>;
```

Pole struktur

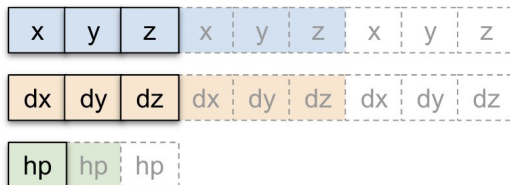
pos (pozice) vel (rychlost) hp (zdraví) + padding



## ● Struct of Arrays

```
namespace SOA {  
struct Beasts {  
    explicit Beasts(size_t size)  
        : pos(size), vel(size), hp(size) {}  
    std::vector<Point3D> pos;  
    std::vector<Point3D> vel;  
    std::vector<HealthPoints> hp;  
};
```

Struktura polí





## ● Array of Structs

```
namespace AOS {  
struct Beast {  
    Point3D pos{};  
    Point3D vel{};  
    HealthPoints hp{};  
};  
using Beasts = std::vector<Beast>;
```

Pole struktur

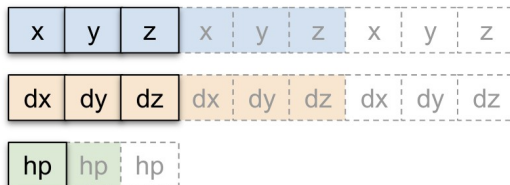
pos (pozice) vel (rychlost) hp (zdraví) + padding



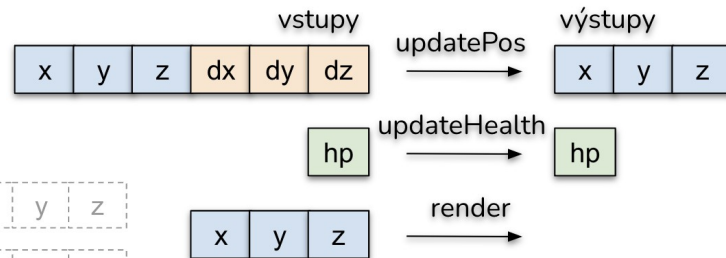
## ● Struct of Arrays

```
namespace SOA {  
struct Beasts {  
    explicit Beasts(size_t size)  
        : pos(size), vel(size), hp(size) {}  
    std::vector<Point3D> pos;  
    std::vector<Point3D> vel;  
    std::vector<HealthPoints> hp;  
};
```

Struktura polí



Transformace



[Godbolt example](#)

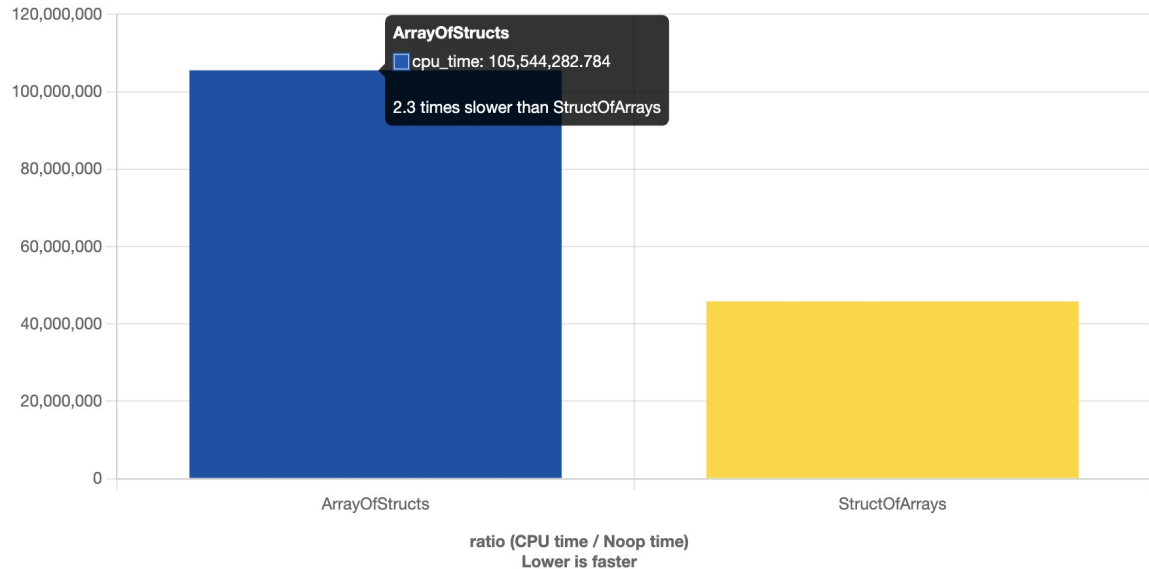
([QuickBench](#))

## ● Array of Structs

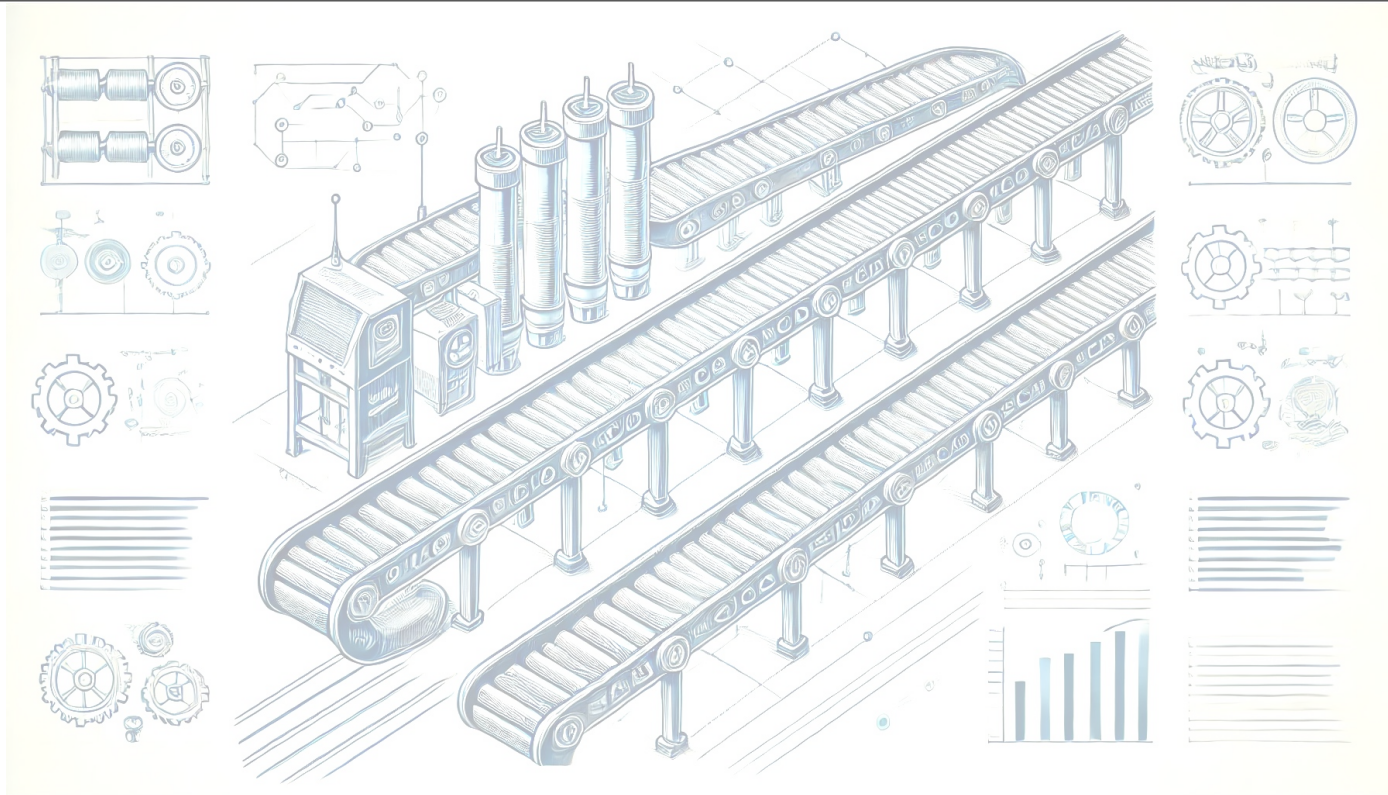
```
namespace AOS {  
struct Beast {  
    Point3D pos{};  
    Point3D vel{};  
    HealthPoints hp{};  
};  
using Beasts = std::vector<Beast>;
```

## ● Struct of Arrays

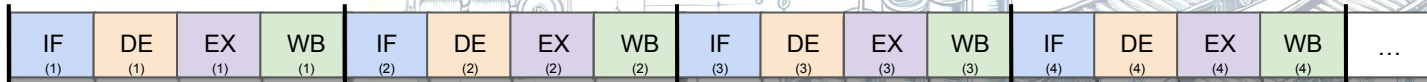
```
namespace SOA {  
struct Beasts {  
    explicit Beasts(size_t size)  
        : pos(size), vel(size), hp(size) {}  
    std::vector<Point3D> pos;  
    std::vector<Point3D> vel;  
    std::vector<HealthPoints> hp;  
};
```



# CPU Pipeline

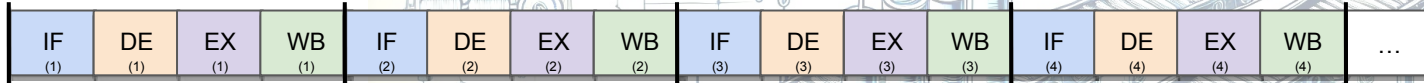


- Sekvenční zpracování

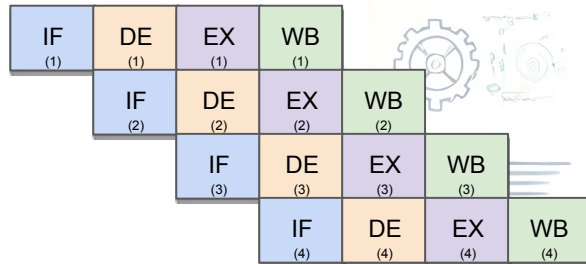


Credit: Jobin Johnson - Branchless programming. Does it really matter?

- Sekvenční zpracování

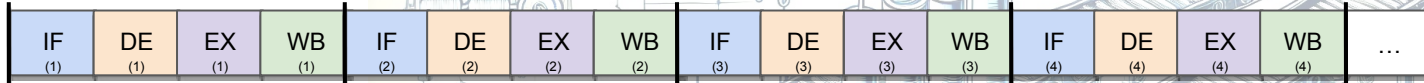


- Pipeline zpracování

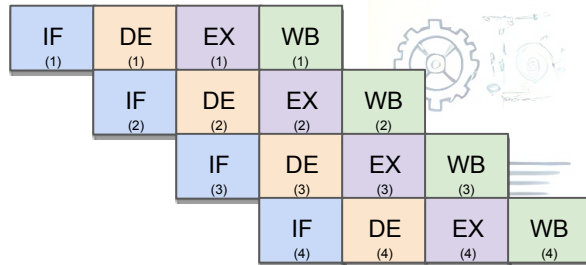


Credit: Jobin Johnson - Branchless programming. Does it really matter?

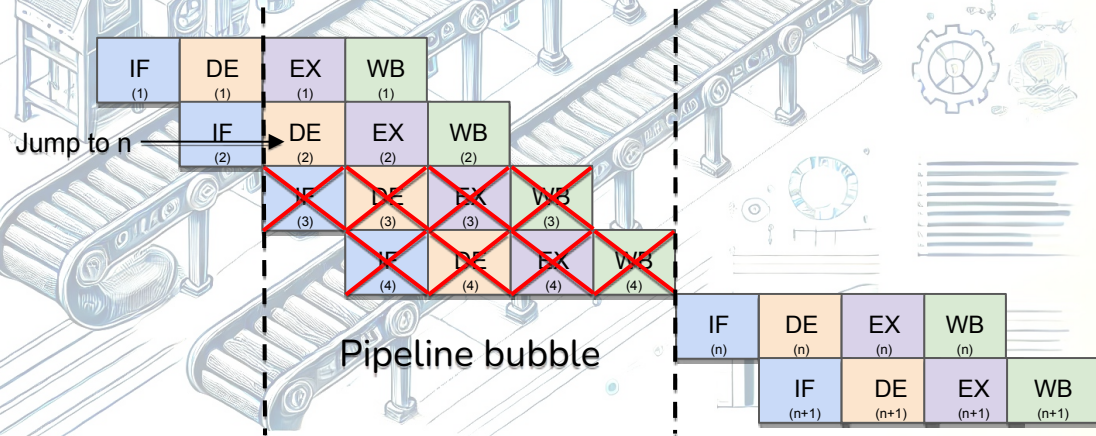
- Sekvenční zpracování



- Pipeline zpracování



- Podmíněný skok (nesprávně predikovaný)



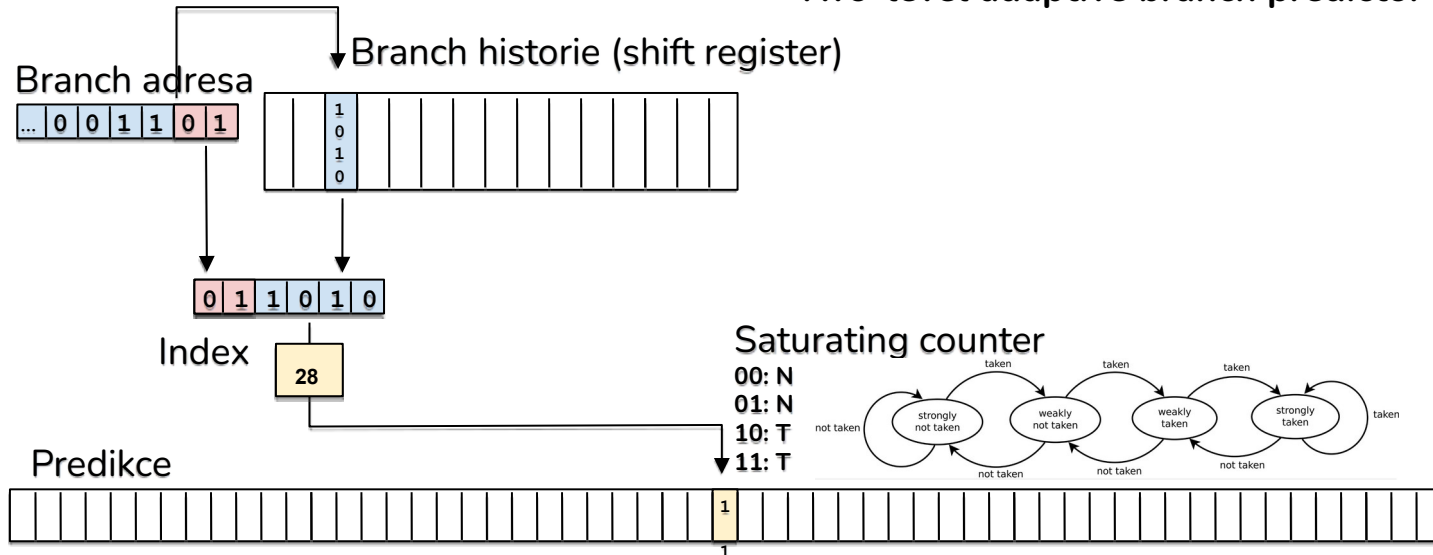
Credit: Jobin Johnson - Branchless programming. Does it really matter?



## ● Branch Predictor

- Obvod v CPU, který předpovídá výsledky podmínek v kódu

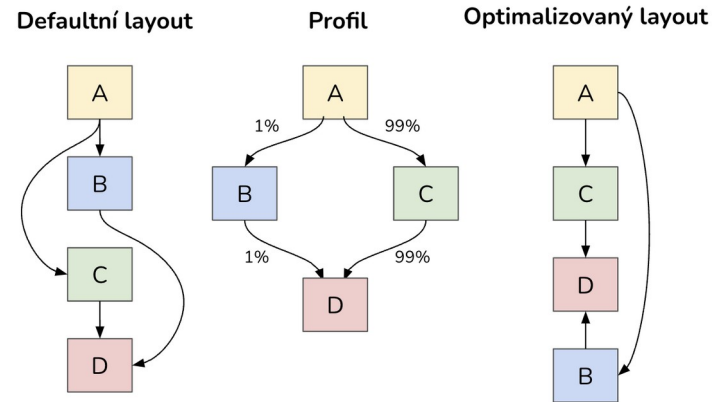
### Two-level adaptive branch predictor



Credit: Dan Luu - Branch prediction



- Branch Predictor
  - Obvod v CPU, který předpovídá výsledky podmínek v kódu
- Profile-guided optimization
  - Optimalizace kódu na základě statistik získaných při běhu programu



- Branch Predictor
  - Obvod v CPU, který předpovídá výsledky podmínek v kódu
- Profile-guided optimization
  - Optimalizace kódu na základě statistik získaných při běhu programu
- Branch Latency

```
1. def handle_trade(trade):  
2.     if trade.size > threshold:  
3.         send_order_triggered_by_trade(trade)
```

- Branch Predictor
  - Obvod v CPU, který předpovídá výsledky podmínek v kódu
- Profile-guided optimization
  - Optimalizace kódu na základě statistik získaných při běhu programu
- Branch Latency
  - CPU i překladače občas pracují “proti nám”
  - Řešením je použití specializovaného HW (FPGA, ...)

# Branchless Programming



# Branchless Programming

```
>>> def side_to_sign(isSell : bool) -> int:  
...     return 1 if isSell else -1  
...  
>>> side_to_sign(True)  
1  
>>> side_to_sign(False)  
-1
```



```
>>> def side_to_sign(isSell : bool) -> int:  
...     return 2 * isSell - 1  
...  
>>> side_to_sign(True)  
1  
>>> side_to_sign(False)  
-1
```

# Branchless Programming

```
>>> def side_to_sign(isSell : bool) -> int:
...     return 1 if isSell else -1
...
>>> side_to_sign(True)
1
>>> side_to_sign(False)
-1
```



```
>>> def side_to_sign(isSell : bool) -> int:
...     return 2 * isSell - 1
...
>>> side_to_sign(True)
1
>>> side_to_sign(False)
-1
```

```
>>> def get_side(multiplier: float, isSell : bool) -> bool:
...     return not isSell if multiplier >= 0.0 else isSell
...
>>> get_side(-0.5, True)
True
>>> get_side(0.5, True)
False
>>> get_side(-0.5, False)
False
>>> get_side(0.5, False)
True
```



```
>>> def get_side(multiplier: float, isSell : bool) -> bool:
...     return (multiplier >= 0.0) ^ isSell
...
>>> get_side(-0.5, True)
True
>>> get_side(0.5, True)
False
>>> get_side(-0.5, False)
False
>>> get_side(0.5, False)
True
```

# Branchless Programming

```
>>> def side_to_sign(isSell : bool) -> int:
...     return 1 if isSell else -1
...
>>> side_to_sign(True)
1
>>> side_to_sign(False)
-1
```



```
>>> def side_to_sign(isSell : bool) -> int:
...     return 2 * isSell - 1
...
>>> side_to_sign(True)
1
>>> side_to_sign(False)
-1
```

```
>>> def get_side(multiplier: float, isSell : bool) -> bool:
...     return not isSell if multiplier >= 0.0 else isSell
...
>>> get_side(-0.5, True)
True
>>> get_side(0.5, True)
False
>>> get_side(-0.5, False)
False
>>> get_side(0.5, False)
True
```



```
>>> def get_side(multiplier: float, isSell : bool) -> bool:
...     return (multiplier >= 0.0) ^ isSell
...
>>> get_side(-0.5, True)
True
>>> get_side(0.5, True)
False
>>> get_side(-0.5, False)
False
>>> get_side(0.5, False)
True
```

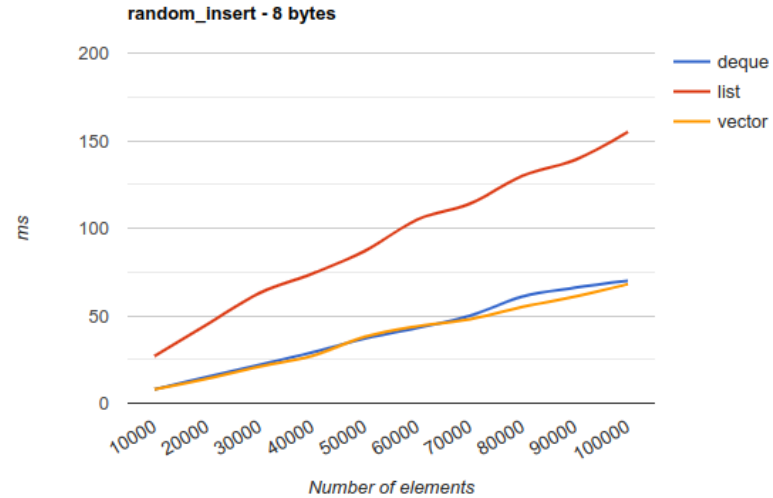
[Godbolt example](#)

[Godbolt example \(llvm-mca & flow-graph\)](#)

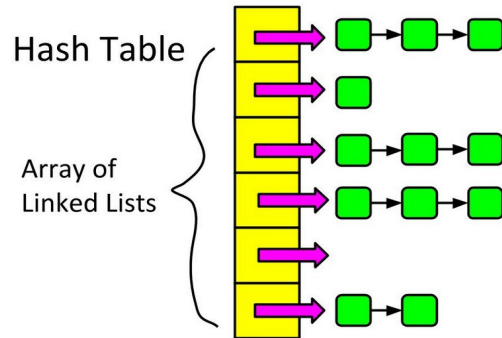




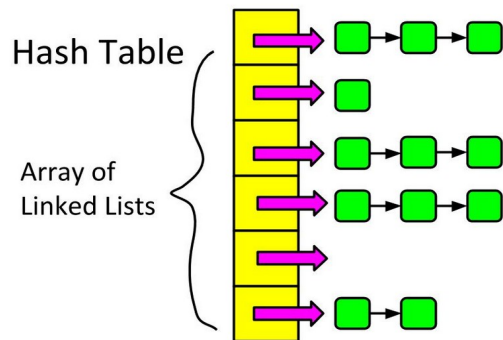
- Pole
  - Naprostá většina datových struktur



- Pole
  - Naprostá většina datových struktur
- Hashovací tabulky



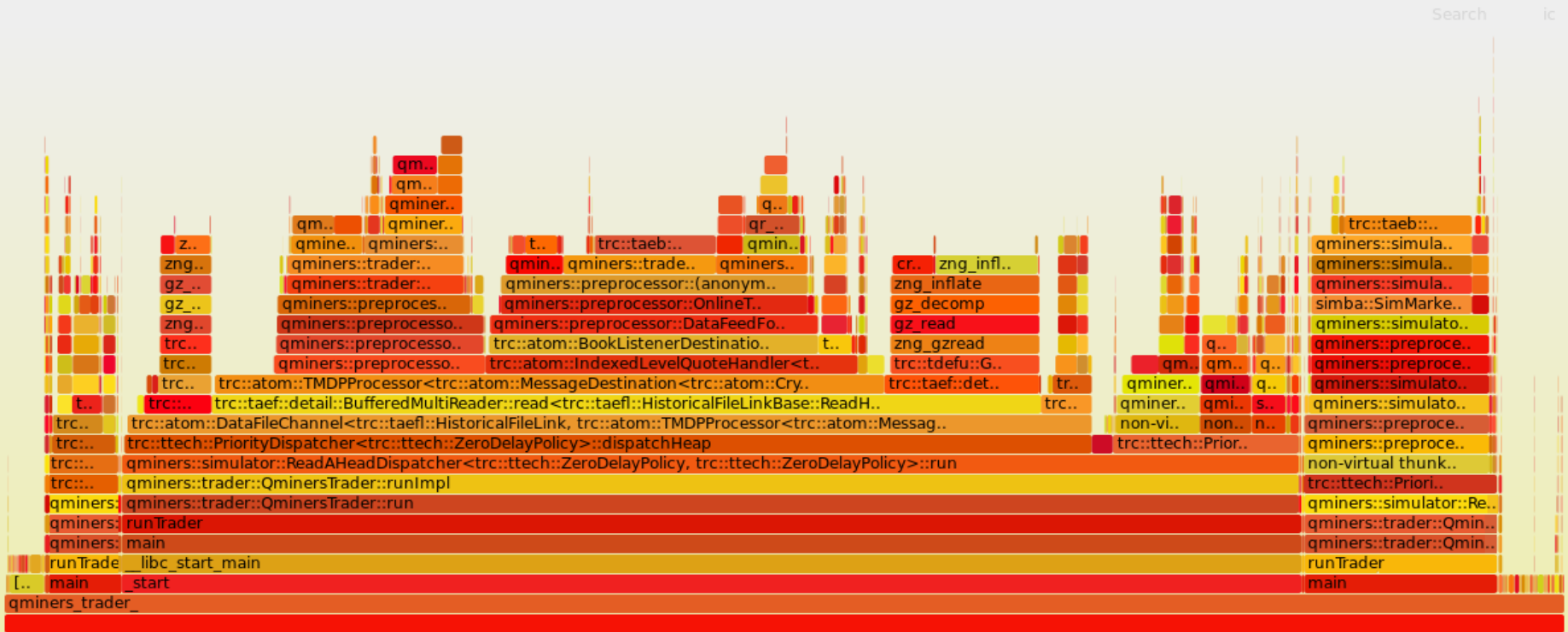
- Pole
  - Naprostá většina datových struktur
- Hashovací tabulky
  - Např. kniha limitních objednávek



Linux perf (FlameGraph visualization)



# Linux perf (FlameGraph visualization)





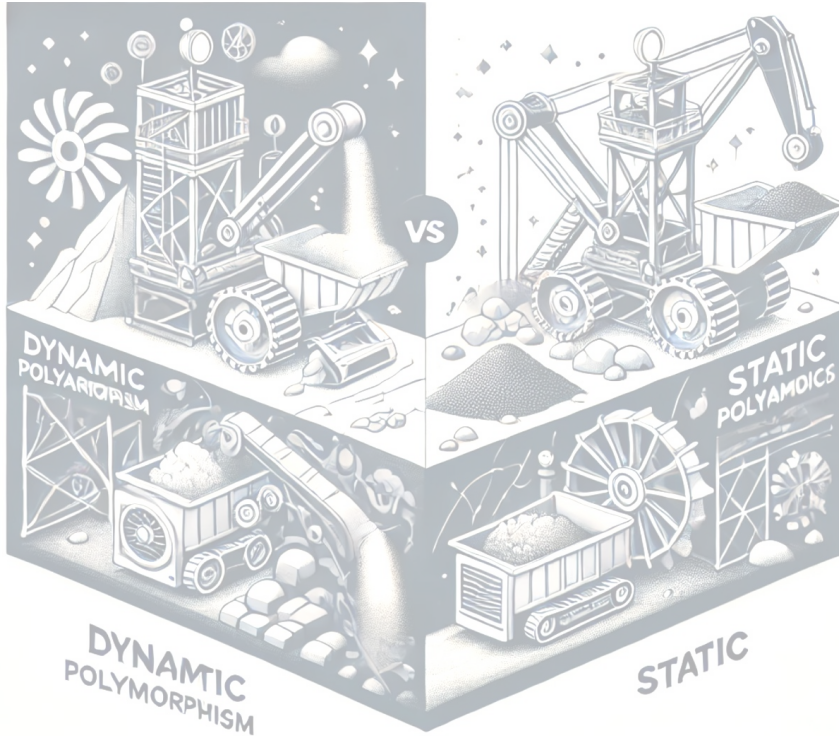








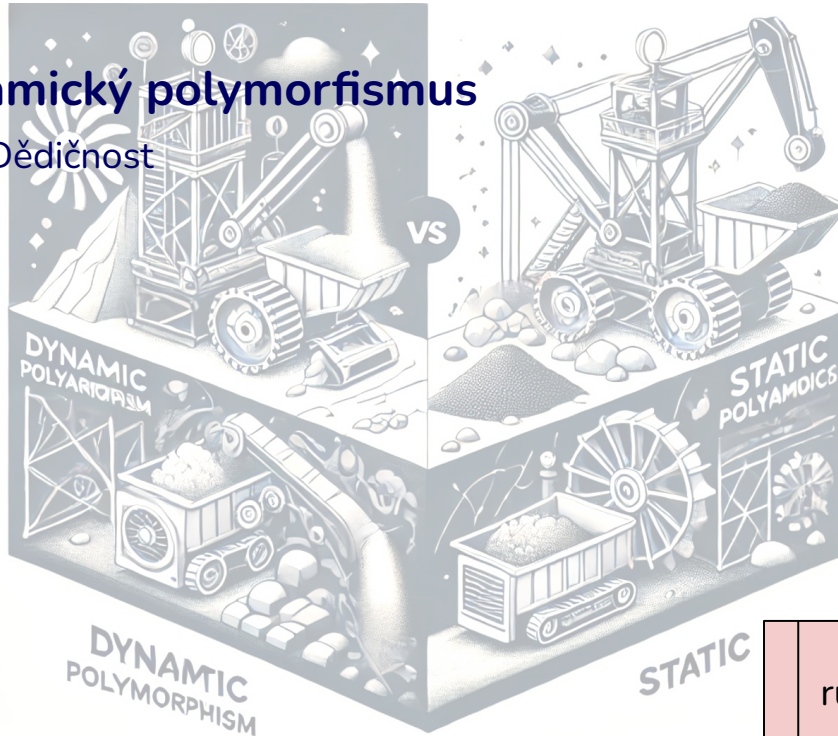
# Dynamic / Static Polymorphism



# Dynamic / Static Polymorphism

- **Dynamický polymorfismus**

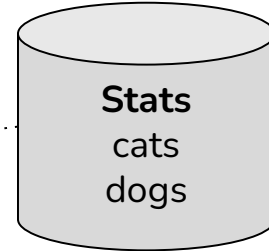
- Dědičnost



Animals



runAll  
(animals, stats)



Animal interface  
run(stats)

Cat  
run(stats)

Dog  
run(stats)

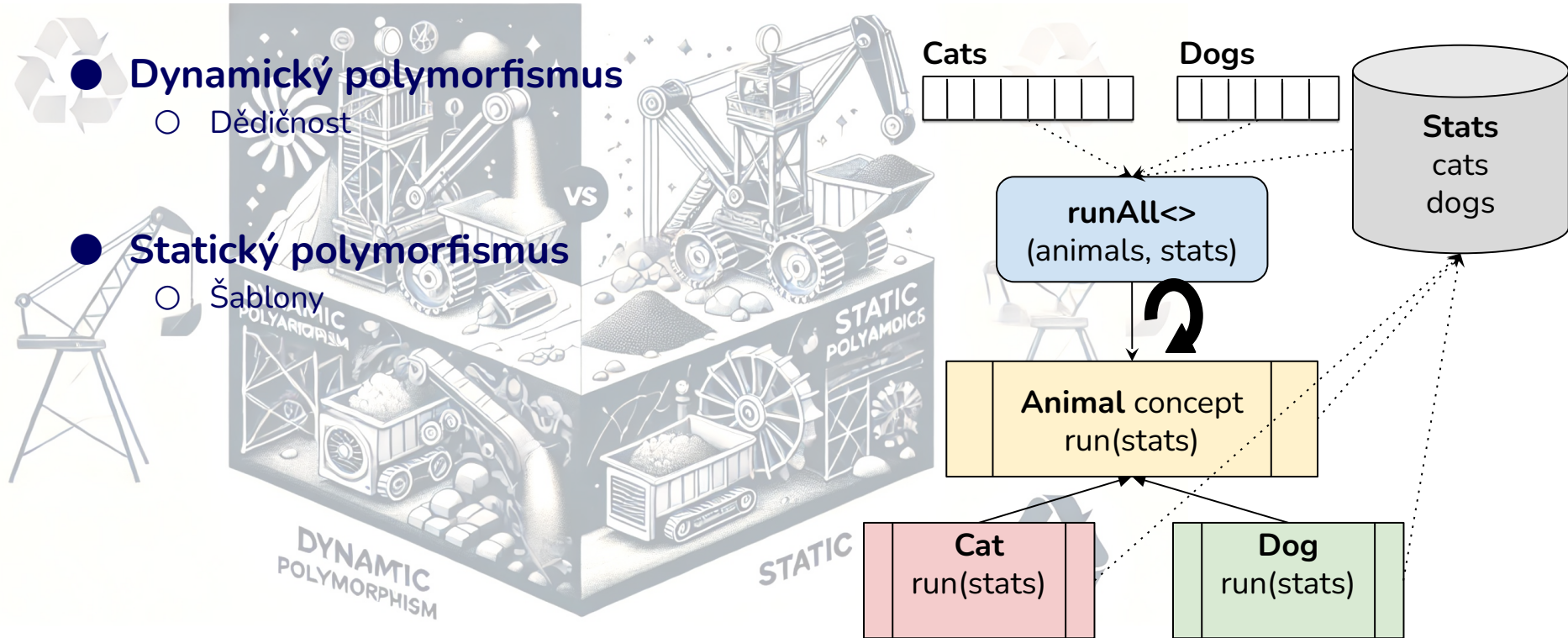
# Dynamic / Static Polymorphism

- **Dynamický polymorfismus**

- Dědičnost

- **Statický polymorfismus**

- Šablony



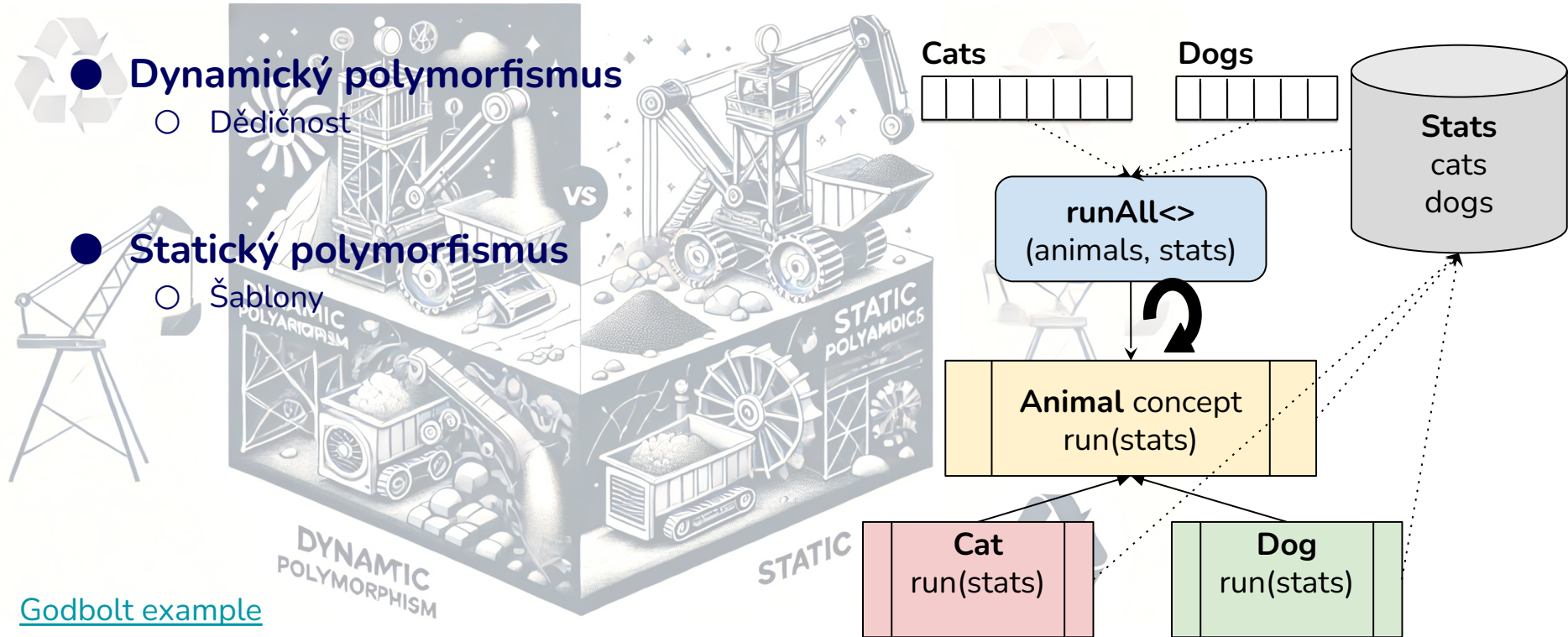
# Dynamic / Static Polymorphism

- **Dynamický polymorfismus**

- Dědičnost

- **Statický polymorfismus**

- Šablony



[Godbolt example](#)

# Efficiency vs. Effectiveness



## ● Efficiency - účinnost

- Dosahujeme cílů s minimálním plýtváním zdroji?
- Kolik prostředků potřebujeme k tomu, co děláme?
- Používáme “the right tool for the job”?



## ● Efficiency - účinnost

- Dosahujeme cílů s minimálním plýtváním zdroji?
- Kolik prostředků potřebujeme k tomu, co děláme?
- Používáme “the right tool for the job”?

## ● Effectiveness - účelnost

- Pracujeme na těch správných věcech?
- Neděláme něco zbytečně složitě?
- Opravdu vše, co děláme, plní svůj účel?





# Summary



- Vysvětlení pojmů na příkladech
  - Latence vs. Propustnost
  - Memory hierarchy a Data oriented design
  - Branch Predictor a Branchless programming
  - Dynamický vs. Statický Polymorfismus
  - Účinnost vs. Účelnost
- Složitosti dnešního HW ... nelze spoléhat na ...
  - ... to, že CPU se bude chovat adekvátně pro váš use-case
  - ... zastaralé poučky ( $O(N)$  lepší než  $O(1)$  s vysokým faktorem)
  - ... optimalizace překladače (je třeba mít intuici, jak HW funguje)
  - ... vlastní intuici (je třeba měřit)
  - ... jeden druh měření (je třeba kombinovat různé techniky)



- Qminers ... <https://qminers.com>

- Videos

- Scott Meyers: Cpu Caches and Why You Care ... <https://www.youtube.com/watch?v=VDIkqP4JbkE>
- Mike Acton: Data-Oriented Design and C++ ... <https://www.youtube.com/watch?v=rX0ItVEVjHc>
- Matt Godbolt: Compiler Explorer 2023: What's New? ... [https://www.youtube.com/watch?v=EyOH79z\\_pco](https://www.youtube.com/watch?v=EyOH79z_pco)
- Fedor Pikus: Branchless Programming in C++ ... <https://www.youtube.com/watch?v=q-WPhYREFik>
- Fedor Pikus: C++ Type Erasure Demystified ... [https://www.youtube.com/watch?v=p-qaf6OS\\_f4](https://www.youtube.com/watch?v=p-qaf6OS_f4)
- Mathieu Ropert: Data Oriented Design and ECS Explained ... <https://www.youtube.com/watch?v=xm4AQj5PHT4>

- Tools

- Compiler explorer ... <https://godbolt.org>
- Valgrind ... <https://valgrind.org/>
- Linux perf ... <https://perfwiki.github.io/main>
- Flamegraph visualization ... <https://github.com/brendangregg/FlameGraph>
- Optick Profiler for games ... <https://optick.dev/>
- Intel VTune Profiler ... <https://www.intel.com/content/www/us/en/developer/tools/oneapi/rtune-profiler.html>

- Articles

- Dan Luu: Branch prediction ... <https://danluu.com/branch-prediction/>
- Ulrich Drepper: What every programmer should know about memory ... <https://people.freebsd.org/~lstewart/articles/cpumemory.pdf>
- Jobin Johnson: Branchless programming. Does it really matter? ... <https://dev.to/jobinjohnson/branchless-programming-does-it-really-matter-2014>
- PACMan: Prefetch-Aware Cache Management for high performance caching ... <https://ieeexplore.ieee.org/document/7851493>
- C++ Tutorial: Intro to Hash Tables ... <https://pumpkinprogrammerdotcom4.wordpress.com/2014/06/21/c-tutorial-intro-to-hash-tables>
- C++ benchmark – std::vector vs. std::list vs. std::deque ... <https://baptiste-wicht.com/posts/2012/12/cpp-benchmark-vector-list-deque.html>
- Xaktly - Determinant & Cramer's rule ... <https://xaktly.com/MatrixDeterminant.html>
- Profile Guided Optimization (PGO) – Under the Hood ... <https://devblogs.microsoft.com/cppblog/profile-guided-optimization-pgo-under-the-hood>
- Björn Fahller – Performance of flat maps ... <https://playfulprogramming.blogspot.com/2017/08/performance-of-flat-maps.html>



Source: [ChatGPT](#)

## Děkuji za pozornost



# Bonus: Data Oriented Design



- **Mapa jako pole**

- Klíče jsou tříděné
- Binární nebo sekvenční vyhledávání

## ● Mapa jako pole

- Klíče jsou tříděné
- Binární nebo sekvenční vyhledávání

## ● Pole párů

1	"I"	2	"II"	4	"IV"	5	"V"	6	"VI"	8	"IIX"	9	"IX"	12	"XII"	13	"XIII"	14	"XIV"	15	"XV"	18	"XIIIX"
---	-----	---	------	---	------	---	-----	---	------	---	-------	---	------	----	-------	----	--------	----	-------	----	------	----	---------

## ● Mapa jako pole

- Klíče jsou tříděné
- Binární nebo sekvenční vyhledávání

## ● Pole párů

1	"I"	2	"II"	4	"IV"	5	"V"	6	"VI"	8	"IIX"	9	"IX"	12	"XII"	13	"XIII"	14	"XIV"	15	"XV"	18	"XIIIX"
---	-----	---	------	---	------	---	-----	---	------	---	-------	---	------	----	-------	----	--------	----	-------	----	------	----	---------

## ● Dvě oddělená pole

1	2	4	5	6	8	9	12	13	14	15	18
---	---	---	---	---	---	---	----	----	----	----	----

"I"	"II"	"IV"	"V"	"VI"	"IIX"	"IX"	"XII"	"XIII"	"XIV"	"XV"	"IIXX"
-----	------	------	-----	------	-------	------	-------	--------	-------	------	--------



# Bonus: Branch Prediction



# Bonus: Branch Prediction

```
unsigned long a1 = 0, a2 = 0;
for (size_t i = 0; i < N; ++i) {
    if (b1[i] || b2[i]) {
        a1 += p1[i];
    } else {
        a1 *= p2[i];
    }
}
```

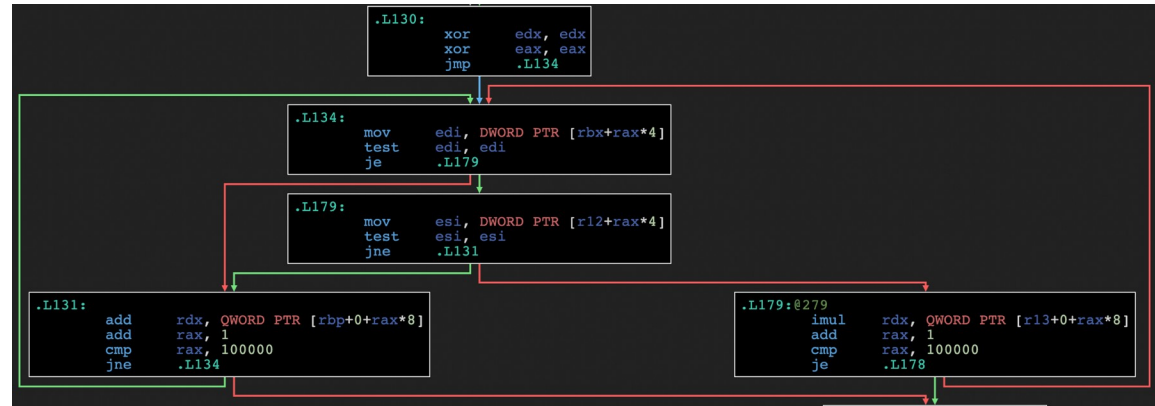
[Godbolt example](#)

([QuickBench](#))

# Bonus: Branch Prediction

```
unsigned long a1 = 0, a2 = 0;
for (size_t i = 0; i < N; ++i) {
    if (b1[i] || b2[i]) {
        a1 += p1[i];
    } else {
        a1 *= p2[i];
    }
}
```

Godbolt example  
([QuickBench](#))



# Bonus: Branch Prediction

```
unsigned long a1 = 0, a2 = 0;
for (size_t i = 0; i < N; ++i) {
    if (b1[i] || b2[i]) {
        a1 += p1[i];
    } else {
        a1 *= p2[i];
    }
}
```

## Godbolt example

(QuickBench)

```
unsigned long a1 = 0, a2 = 0;
for (size_t i = 0; i < N; ++i) {
    if (b1[i] | b2[i]) {
        a1 += p1[i];
    } else {
        a1 *= p2[i];
    }
}
```



Credit: Fedor Pikus: Branchless Programming in C++

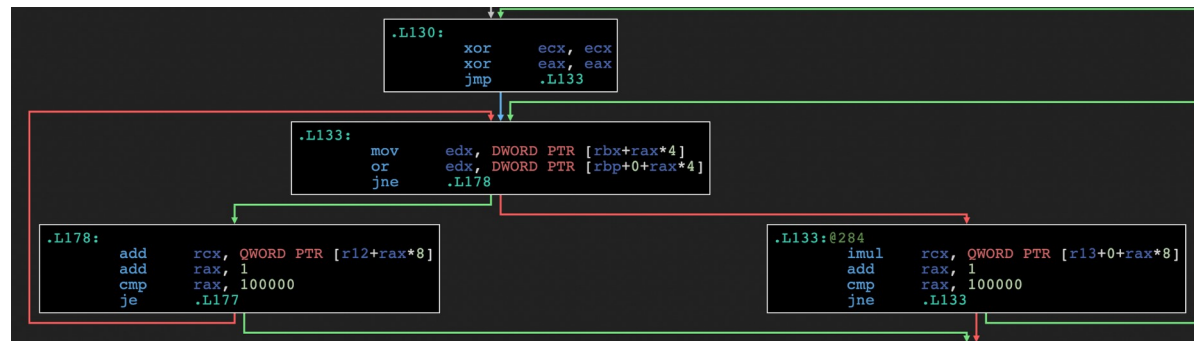
# Bonus: Branch Prediction

```
unsigned long a1 = 0, a2 = 0;
for (size_t i = 0; i < N; ++i) {
    if (b1[i] || b2[i]) {
        a1 += p1[i];
    } else {
        a1 *= p2[i];
    }
}
```

## Godbolt example

(QuickBench)

```
unsigned long a1 = 0, a2 = 0;
for (size_t i = 0; i < N; ++i) {
    if (b1[i] | b2[i]) {
        a1 += p1[i];
    } else {
        a1 *= p2[i];
    }
}
```



Credit: Fedor Pikus: Branchless Programming in C++

# Bonus: Branch Prediction



	Callgrind (#instr)	Perf (time)	Details
<b>Logical OR</b>	134,563,029	0.853 ms	<pre> Benchmark                               Time          CPU           Iterations UserCounters... ----- BranchMisprediction  853273 ns     852790 ns           821 items_per_second=117.262M/s  Performance counter stats for './01-mispredict':        829.819888      task-clock (msec)    #    0.975 CPUs utilized          20          context-switches          #    0.024 K/sec           1          cpu-migrations          #    0.001 K/sec         2035         page-faults              #    0.002 M/sec   2312066437         cycles                    #    2.786 GHz &lt;not supported&gt;    stalled-cycles-frontend &lt;not supported&gt;    stalled-cycles-backend   2235067958         instructions          #    0.97  insns per cycle   353769199          branches                # 426.320 M/sec   46081671          branch-misses         #   13.03% of all branches                     </pre>
<b>Bitwise OR</b>	172,902,709	0.506 ms	<pre> Benchmark                               Time          CPU           Iterations UserCounters... ----- BranchPrediction    506101 ns     502065 ns          1383 items_per_second=199.177M/s  Performance counter stats for './01-predict':     786.186525        task-clock (msec)    #    0.966 CPUs utilized          78          context-switches          #    0.099 K/sec           1          cpu-migrations          #    0.001 K/sec         2035         page-faults              #    0.003 M/sec   2189915718         cycles                    #    2.785 GHz &lt;not supported&gt;    stalled-cycles-frontend &lt;not supported&gt;    stalled-cycles-backend   3870538861         instructions          #    1.77  insns per cycle   475422439          branches                # 604.720 M/sec       85181          branch-misses         #    0.02% of all branches                     </pre>

# Bonus: Branchless Programming



# Bonus: Branchless Programming



```
#include <vector>
#include <math.h>

std::vector<double> existing;
std::vector<double> offset;
std::vector<double> standalone;

double get(int ci, bool applyOffset, bool applyStandalone)
{
    auto res = existing[ci];
    if (applyOffset)
        res += offset[ci];
    if (applyStandalone)
        res += standalone[ci];
    return res;
}
```



# Bonus: Branchless Programming



```
#include <vector>
#include <math.h>

std::vector<double> existing;
std::vector<double> offset;
std::vector<double> standalone;

double get(int ci, bool applyOffset, bool applyStandalone)
{
    auto res = existing[ci];
    if (applyOffset)
        res += offset[ci];
    if (applyStandalone)
        res += standalone[ci];
    return res;
}
```

```
#include <vector>
#include <math.h>

std::vector<double> existing;
std::vector<double> offset;
std::vector<double> standalone;

double get(int ci, bool applyOffset, bool applyStandalone)
{
    return existing[ci] + applyOffset * offset[ci] + applyStandalone * standalone[ci];
}
```

# Bonus: Branchless Programming

```
#include <vector>
#include <math.h>

std::vector<double> existing;
std::vector<double> offset;
std::vector<double> standalone;

double get(int ci, bool applyOffset, bool applyStandalone)
{
    auto res = existing[ci];
    if (applyOffset)
        res += offset[ci];
    if (applyStandalone)
        res += standalone[ci];
    return res;
}
```

```
#include <vector>
#include <math.h>

std::vector<double> existing;
std::vector<double> offset;
std::vector<double> standalone;

double get(int ci, bool applyOffset, bool applyStandalone)
{
    return existing[ci] + applyOffset * offset[ci] + applyStandalone * standalone[ci];
}
```



```
get(int, bool, bool):
    mov     rax, QWORD PTR existing[rip]
    movsx  rdi, edi
    movsd  xmm0, QWORD PTR [rax+rdi*8]
    test   sil, sil
    je     .L5

get(int, bool, bool):%66
    mov     rax, QWORD PTR offset[rip]
    addsd  xmm0, QWORD PTR [rax+rdi*8]

.L5:
    test   dl, dl
    je     .L4

.L5:@11
    mov     rax, QWORD PTR standalone[rip]
    addsd  xmm0, QWORD PTR [rax+rdi*8]

.L4:
    ret
```

# Bonus: Branchless Programming

```
#include <vector>
#include <math.h>

std::vector<double> existing;
std::vector<double> offset;
std::vector<double> standalone;

double get(int ci, bool applyOffset, bool applyStandalone)
{
    auto res = existing[ci];
    if (applyOffset)
        res += offset[ci];
    if (applyStandalone)
        res += standalone[ci];
    return res;
}
```

```
#include <vector>
#include <math.h>

std::vector<double> existing;
std::vector<double> offset;
std::vector<double> standalone;

double get(int ci, bool applyOffset, bool applyStandalone)
{
    return existing[ci] + applyOffset * offset[ci] + applyStandalone * standalone[ci];
}
```

## Godbolt example

(QuickBench #1)

(QuickBench #2)



```
get(int, bool, bool):
    mov     rax, QWORD PTR existing[rip]
    movsx  rdi, edi
    movsd  xmm0, QWORD PTR [rax+rdi*8]
    test   sil, sil
    je     .L5

get(int, bool, bool):@86
    mov     rax, QWORD PTR offset[rip]
    addsd  xmm0, QWORD PTR [rax+rdi*8]

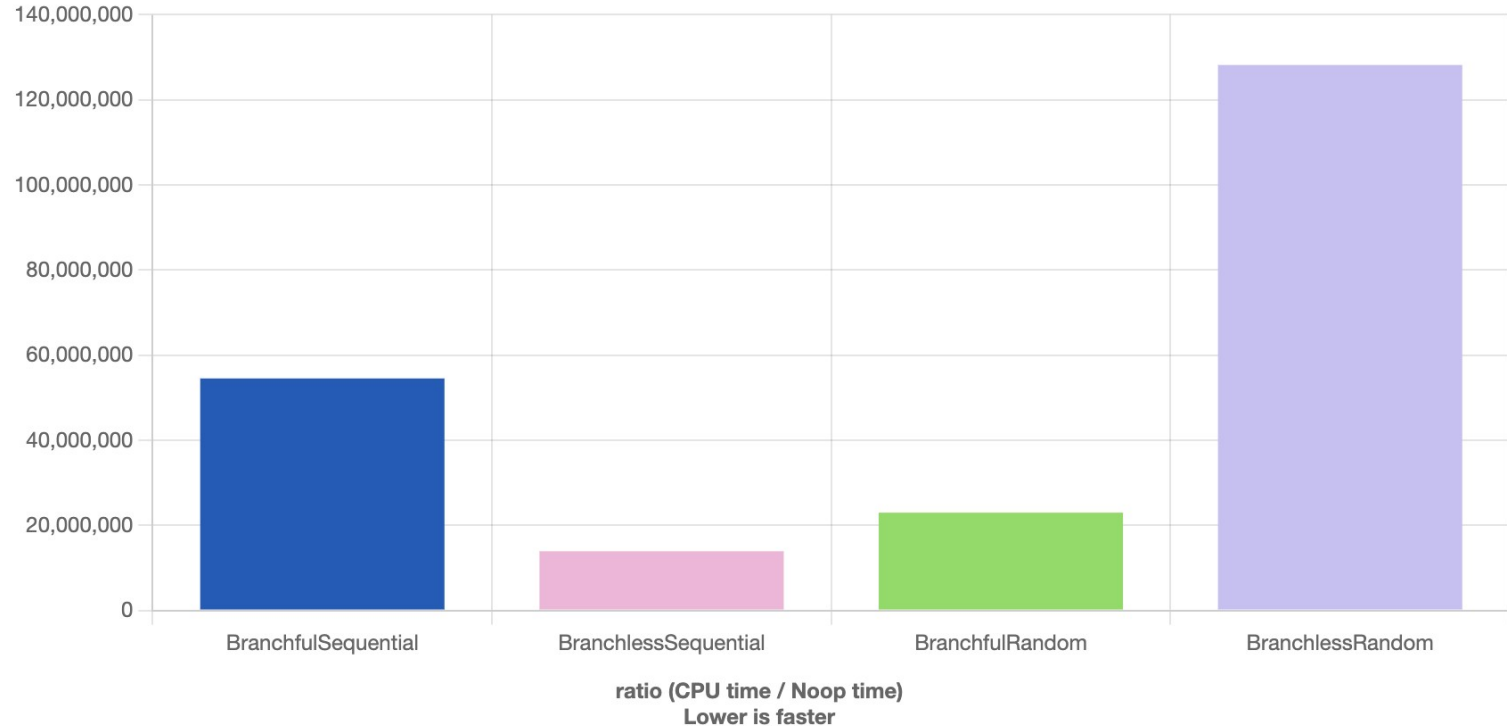
.L5:
    test   dl, dl
    je     .L4

.L5:@11
    mov     rax, QWORD PTR standalone[rip]
    addsd  xmm0, QWORD PTR [rax+rdi*8]

.L4:
    ret
```

```
get(int, bool, bool):
    movsx  rdi, edi
    movzx  esi, sil
    pxor   xmm0, xmm0
    movzx  edx, dl
    mov    rax, QWORD PTR offset[rip]
    cvtsi2sd  xmm0, esi
    pxor   xmm1, xmm1
    cvtsi2sd  xmm1, edx
    mulsd  xmm0, QWORD PTR [rax+rdi*8]
    mov    rax, QWORD PTR existing[rip]
    addsd  xmm0, QWORD PTR [rax+rdi*8]
    mov    rax, QWORD PTR standalone[rip]
    mulsd  xmm1, QWORD PTR [rax+rdi*8]
    addsd  xmm0, xmm1
    ret
```

# Bonus: Branchless Programming

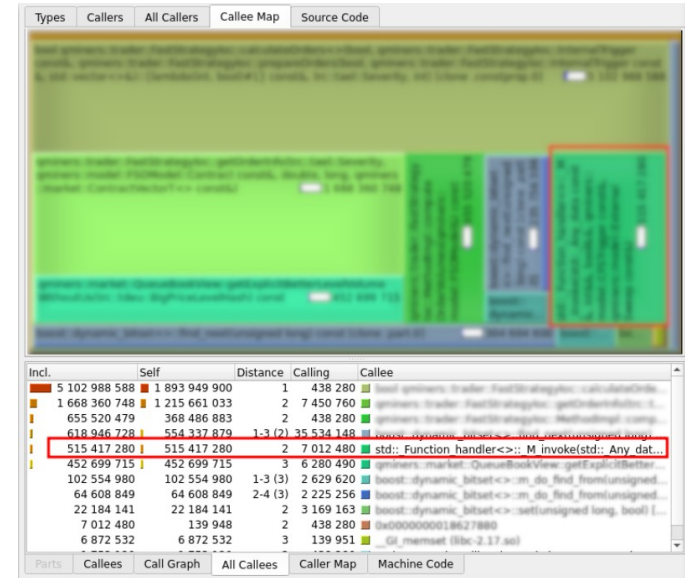
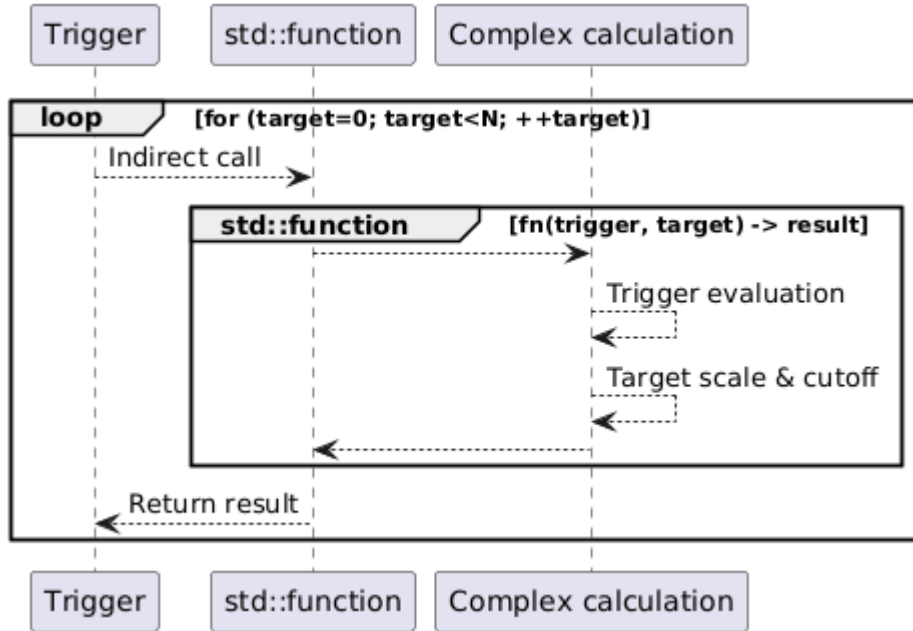


# Bonus: Indirect Calls



- Nepřímé volání
  - Poměrně drahé
    - Branch misprediction, icache miss, optimalizace
    - Snažíme se mu vyhnout (v kritickém kódu)
  - Jen když ho opravdu potřebujeme
    - “inheritance” (*interface*)
    - “type erasure” (*std::function*)
- Správná granularita
  - Kód je třeba dobře strukturovat
  - Měřit
  - Experimentovat

# Bonus: Indirect Calls





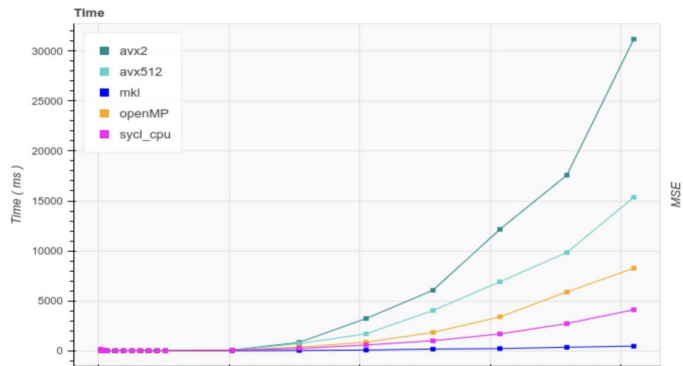


# Bonus: Efficiency vs. Effectiveness



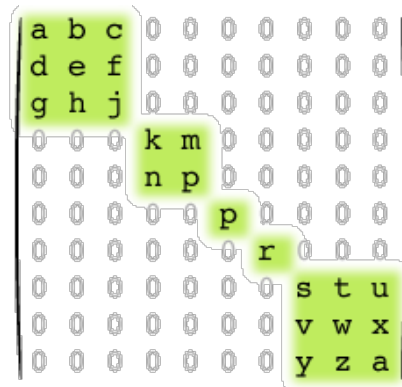
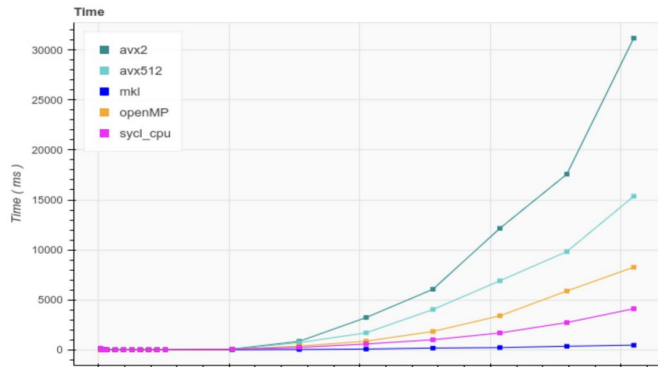
## 1) Lineární algebra

- Použití [MKL](#) (Math Kernel Library)
- Malé matice a vektory, rozklad na [block-diagonal-matrix](#)



## 1) Lineární algebra

- a) Použití [MKL](#) (Math Kernel Library)
- b) Malé matice a vektory, rozklad na [block-diagonal-matrix](#)



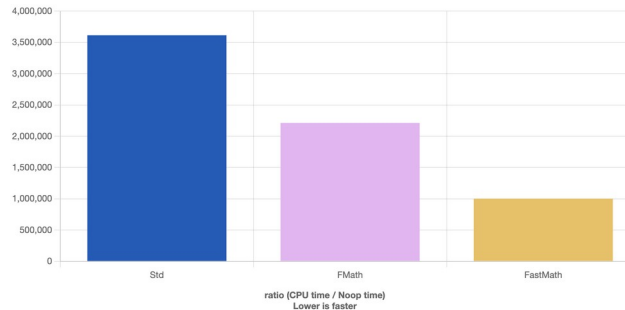
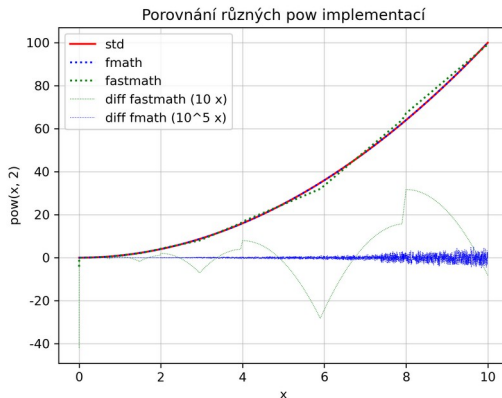
# Bonus: Efficiency vs. Effectiveness

## 1) Lineární algebra

- a) Použití [MKL](#) (Math Kernel Library)
- b) Malé matice a vektory, rozklad na [block-diagonal-matrix](#)

## 2) Exponenciální funkce

- a) Použití knihoven ([std](#), [fmath](#), [fastmath](#), ...)
- b) Je nutné volat  $\text{pow}(x, \alpha)$ , když  $\alpha == 1.05$ ?



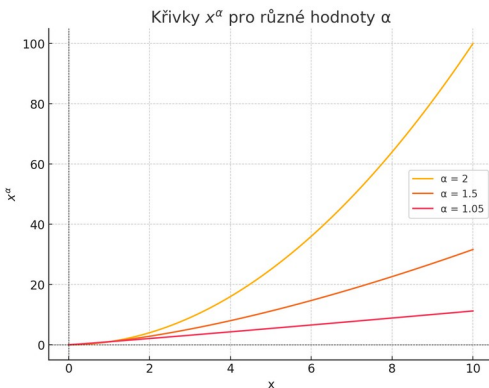
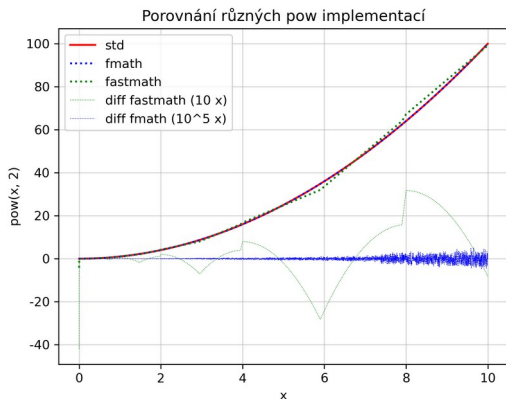
[Godbolt example](#)

## 1) Lineární algebra

- a) Použití [MKL](#) (Math Kernel Library)
- b) Malé matice a vektory, rozklad na [block-diagonal-matrix](#)

## 2) Exponenciální funkce

- a) Použití knihoven ([std](#), [fmath](#), [fastmath](#), ...)
- b) Je nutné volat `pow(x, alpha)`, když `alpha == 1.05`?



[Godbolt example](#)