Sekvenční architektury II

Zpracování instrukcí

Jak zvýšit výkon CPU

- zkrátit čas nutný ke zpracování 1 instrukce
  - urychlit časovač (Timer) = zvýšení taktu
    - to je technicky velmi náročné, poslední dobou se frekvence CPU nemění $\approx 3GHz$
    - zvyšování frekvence vede k velkému nárůstu produkovaného tepla
    - i samotné zvyšování frekvence vyžaduje zásah do architektury
  - použít tzv. *pipelining* = jistá forma paralelizace
- provádět více operací najednou
  - tím již dostáváme paralelní systém
  - ale tzv. *superskalární zpracování* patří mezi implicitně paralelní systémy

Definition

Je-li některý systém schopný paralelně zpracovávat kód psaný pro sekvenční systém, mluvíme o **implicitně paralelním systému**.

Přehled procesorů firmy Intel

Název	Architektura	Rok uvedení
8086,80186,80286	86	1978
80386,80386SX,80386DX	80386	1985
80486,80486SX,80486DX	80486	1989
Pentium, Pentium MMX	P5	1993
Pentium Pro,Pentium 2,3,Celeron	P6	1995
Pentium 4	Netburst	2000
Intel Core2	Core arch.	2006
Core i3, i5, i7, Xeon	Nehalem	2010
Core i3, i5, i7, Xeon	Sandy Bridge	2011
Core i3, i5, i7, Xeon	Ivy Bridge	2012
Core i3, i5, i7, Xeon	Haswell	2013
Core i3, i5, i7, Xeon	Skylake	2015
Core i3, i5, i7, Xeon	Kabylake	2016
Core i3, i5, i7, Xeon	Cannonlake	2017

Přehled procesorů firmy AMD

Název	Architektura	Rok uvedení
Am386, Am486, Am5x86	Amx86	1979
K5	K5	1995
K6, K6-2, K6-III	K6	1997
Athlon, Duron, Sempron	K7	1999
Athlon 64, Opteron, Sempron	K8 core	2003
Phenom, Athlon, Opteron	K10	2007
FX-8xxx, FX-6xxx, FX-4xxx	Bulldozer	2011
	Bobcat	
	Jaguar	
	Zen	2017

Pipelining I.

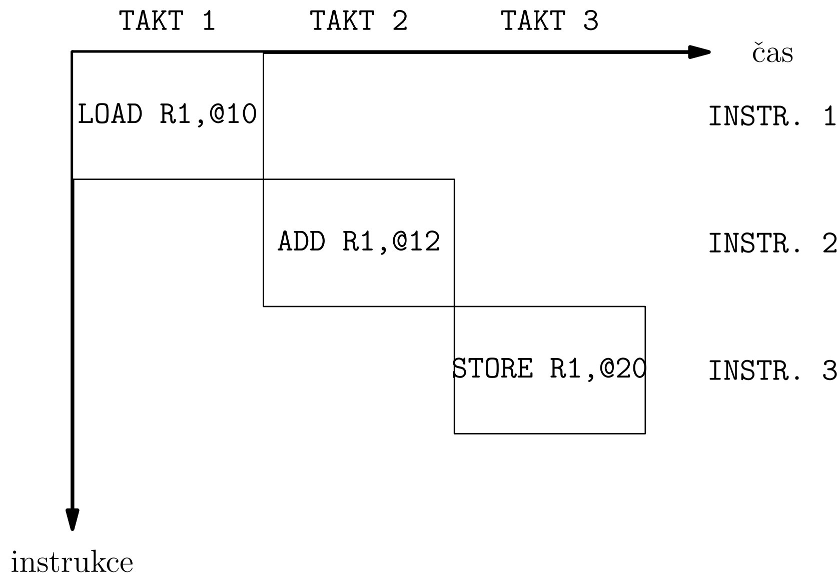
Procesor zpracovává instrukce pomocí pipeliningu:

- umožňuje urychlit zpracování instrukcí
- dovolí taktování na vyšší frekvence

Vezměme si následující kód:

```
LOAD  R1, @10 # nacti do reg. 1 hodnotu z adresy 10
ADD   R1, @12 # pricti do reg. 1 hodnotu a adresy 12
STORE R1, @20 # zapis reg. 1 na adresu 20
```

Pipelining III.

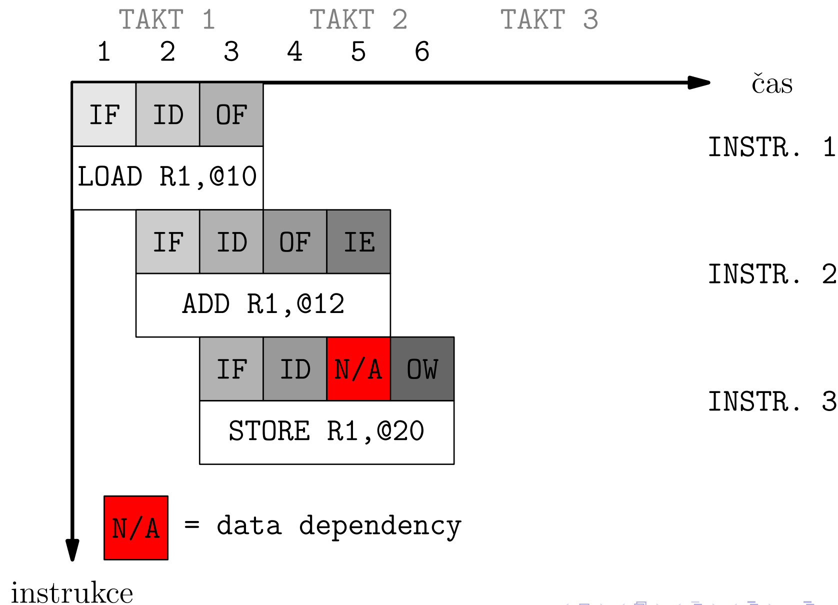


Pipelining IV.

PIPELINING se podobá výrobní lince

- zpracování instrukce se rozdělí na více kroků, např.
  - načtení instrukce (instruction fetch) - IF
  - dekódování instrukce (instruction decode) - ID
  - načtení operandu (operand fetch) - OE
  - provedení instrukce (instruction execution) - IE
  - zápis operandu (operand write) - OW
- tyto kroky jsou jednodušší na provedení
- díky tomu lze zkrátit délku jednoho taktu - ten nyní odpovídá jednomu kroku zpracování
- provádění lze zřetězit

Pipelining V.



Pipelining VI.

Architektura	Délka pipeline
P5	5
P6	10-14
Netburst	20-31
Core	14
Nehalem–Haswell	14-19

Pipelining VII.

Pipelining se potýká se třemi typy závislostí, které snižují jeho efektivitu:

- **datová závislost** - data dependency
  - nastává ve chvíli, kdy jedna instrukce potřebuje data, která ještě nejsou spočtena
- **závislost na zdrojích** - resource dependency
  - v situaci, kdy dvě instrukce chtějí přistupovat např. na stejné místo v paměti nebo do stejného registru
- **závislost na podmínce** - branch dependency
  - v situaci, kdy není znám výsledek výpočtu, jenž může ovlivnit, jakým způsobem bude kód dále prováděn (např. nejsou data k vyhodnocení podmíněného skoku)
  - udává se, že v průměru se vyskytuje jedna podmínka na každých 5-6 instrukcí

Pipelining VIII.

Problémy s datovou závislostí a závislostmi na zdrojích se řeší pomocí:

- **provádění instrukcí mimo pořadí** - *out of order execution*
  - procesor přehazuje pořadí zpracování instrukcí tak, aby se vyhnul zmiňovaným závislostem
  - může to provádět i překladač při různých optimalizacích
  - většinou nepomáhá u závislostech na podmínce
- **přejmenování registrů** – *registers renaming*
  - procesor se může rozhodnout, že změní uložení proměnných v registrech, pokud to pomůže lepšímu zpracování kódu

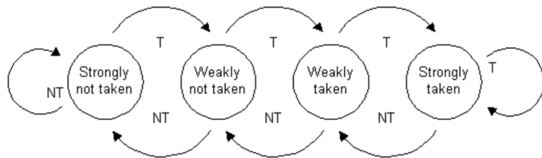
Předvídání podmínek

Problémy se závislostí na podmínce se řeší pomocí:

- **předvídání výsledků podmínek** - *speculative execution, branch prediction*
  - procesor se pokusí uhodnout, která větev programu má větší šanci, že bude prováděna
  - tu pak začne pomocí pipeline zpracovávat jěště před vyřešením samotné podmínky
  - to má překvapivě velkou úspěšnost až 99%
  - pokud předvídání selže, je nutné začít zpracovávat druhou větev
  - to si vyžaduje vyprázdnění celé pipeline, což vede k velkému zpomalení
    - na architektuře Nehalem odhadem 20 taktů
  - to je důvod, proč nelze dělat pipeliny příliš dlouhé

Předvídání podmínek

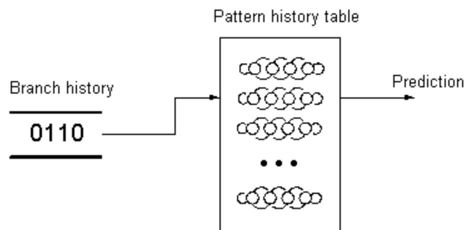
- k tomu, aby procesor dokázal odhadnout výsledek podmíněného skoku, používá set associative cache zvanou *Branch Target Buffer*
- velikost se nezná, ale odhaduje se na 4kB (P5) a 8-16kB (Sandy Bridge)
- cache používá jako klíč adresu cíle skoku
- uložená hodnota odpovídá pravděpodobnosti, že skok bude proveden (*taken*) nebo ne (*not taken*) a zpracuje se hned následující instrukce



Zdroj: `www.agner.org/optimize/optimizing_cpp.pdf`

Předvídání podmínek

- procesor si navíc pamatuje kratkou historii toho, jak podmíněné skoky dopadly v posledních n případech
- za tím účelem se používá jednoduchá metoda vyvinutá T.Y.Yehem a Y.N.Pattem

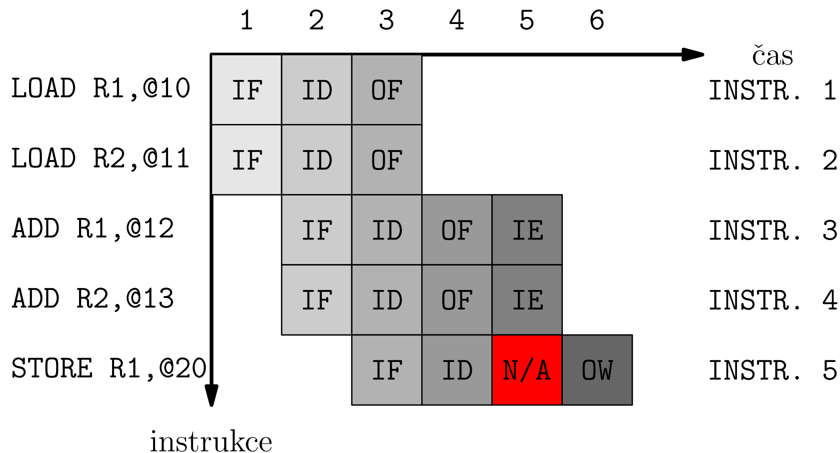


Zdroj: `www.agner.org/optimize/optimizing_cpp.pdf`

Superskalární zpracování I.

- jsou-li dvě instrukce na sobě zcela nezávislé, je možné je zpracovat obě současně
- lze tak využít dvě nebo více pipeline
- k efektivnímu využití je potřeba mít instrukce dobře setříděné
  - to lze dělat během překladu - překladač
  - nebo za chodu programu - procesor
- Intel zavedl superskalární zpracování v procesorech P5 Pentium

Superskalární zpracování II.



Optimalizace na zpracování podmínek

Pro optimální využití pipeline je nutné minimalizovat vyšetřování podmínek, obzvláště těch, které jsou špatně předvídatelné

Příklad: Budeme opakovaně provádět následující kód:

```
1  void performTest( const int* data,
2                    const int size,
3                    int& a )
4  {
5     a = 0;
6     for( int i = 0; i < size; i++ )
7     {
8        if( data[ i ] == 1 )
9           a++;
10       if( data[ i ] == 0 )
11          a = 0;
12       if( data[ i ] == -1 )
13          a--;
14    }
15 }
```

Optimalizace na zpracování podmínek

- pole `data` budeme naplňovat prodlužijícím se opakujicím náhodným vzorem hodnot `-1,0,1`

```
1  void setupTest( int* data,
2                  const int size,
3                  const int patternLength )
4  {
5     int* pattern = new int[ patternLength ];
6     srand( 0 );
7     for( int i = 0; i < patternLength; i++ )
8        pattern[ i ] = rand() % 3 - 1;
9     for( int i = 0; i < size; i++ )
10    {
11       data[ i ] = pattern[ i % patternLength ];
12    }
13    delete[] pattern;
14 }
```

Optimalizace na zpracování podmínek

Výsledky:

Délka vzoru	Neúspěšnost	
	bez `-O3`	s `-O3`
1	0.0006%	0.0006%
2	0.0005%	0.0004%
4	0.0010%	0.0005%
8	3.9955%	3.0589%
16	7.1143%	4.6074%
32	5.1034%	1.4232%
64	8.0148%	2.2992%
128	10.9170%	2.5160%
256	15.0898%	3.4148%
512	16.9543%	11.1091%
1024	17.8589%	12.1402%
2048	18.7427%	14.9048%
4096	18.6398%	17.6160%
8192	18.3994%	19.6302%
16384	18.3008%	21.0191%
32768	18.3611%	21.6142%
65536	18.3970%	21.9362%
131072	18.3810%	22.0045%
262144	18.3722%	22.0538%
524288	18.3735%	22.0417%
1048576	18.3635%	22.0176%
2097152	18.3620%	22.0096%

- vidíme, že s prodlužijícím se vzorem opakování klesá efektivita předvídání podmínek

Optimalizace na zpracování podmínek

- nesmíme také zapomenout, že každý cyklus v sobě obsahuje podmínku, jež se musí řešit v každé smyčce
- její výsledek bude vždy stejný až na poslední průchod, takže bude předvídána s velkou efektivitou (alespoň u delších cyklů)
- i tak se ale vyplatí se této podmínky v cyklu zbavit pomocí tzv. rozbalení smyčky *loop unrolling*

Optimalizace na zpracování podmínek

```
 1  void performUnrolledTest( const int* data,
 2                            const int size,
 3                            int& a )
 4  {
 5     a = 0;
 6     for( int i = 0; i < size; i += 4 )
 7     {
 8        if( data[ i ] == 1 )
 9           a++;
10        if( data[ i ] == 0 )
11           a = 0;
12        if( data[ i ] == -1 )
13           a--;
14
15        if( data[ i + 1 ] == 1 )
16           a++;
17        if( data[ i + 1 ] == 0 )
18           a = 0;
19        if( data[ i + 1 ] == -1 )
20           a--;
21
22        if( data[ i + 2 ] == 1 )
23           a++;
24        if( data[ i + 2 ] == 0 )
25           a = 0;
26        if( data[ i + 2 ] == -1 )
27           a--;
28
29        if( data[ i + 3 ] == 1 )
30           a++;
31        if( data[ i + 3 ] == 0 )
32           a = 0;
33        if( data[ i + 3 ] == -1 )
34           a--;
35     }
36  }
```

Optimalizace na zpracování podmínek

Výsledky (neúspěšně předvídané podmínky):

Délka vzoru	bez rozbalení		s rozbalením	
	bez `-O3`	s `-O3`	bez `-O3`	s `-O3`
1	0.0006%	0.0006%	0.0005%	0.0006%
2	0.0005%	0.0004%	0.0005%	0.0005%
4	0.0010%	0.0005%	0.0005%	0.0004%
8	3.9955%	3.0589%	0.0012%	0.0009%
16	7.1143%	4.6074%	0.0014%	0.0011%
32	5.1034%	1.4232%	0.0021%	0.0017%
64	8.0148%	2.2992%	2.0792%	0.3154%
128	10.9170%	2.5160%	3.9394%	0.0028%
256	15.0898%	3.4148%	7.3365%	1.0891%
512	16.9543%	11.1091%	11.9075%	2.3585%
1024	17.8589%	12.1402%	16.8415%	4.1920%
2048	18.7427%	14.9048%	20.7197%	6.7165%
4096	18.6398%	17.6160%	22.6760%	10.5239%
8192	18.3994%	19.6302%	22.8864%	15.7348%
16384	18.3008%	21.0191%	22.8314%	20.9794%
32768	18.3611%	21.6142%	22.8662%	25.6041%
65536	18.3970%	21.9362%	22.8915%	28.3501%
131072	18.3810%	22.0045%	22.8386%	29.7677%
262144	18.3722%	22.0538%	22.8021%	30.1786%
524288	18.3735%	22.0417%	22.7777%	30.3599%
1048576	18.3635%	22.0176%	22.7585%	30.3551%
2097152	18.3620%	22.0096%	22.7403%	30.2972%

Optimalizace na zpracování podmínek

- rozbalení smyček opravdu pomáhá
- rozbalení smyček umí provádět i překladač
- nevýhodou je, že se zvětšuje kód, což může zatěžovat instrukční cache

Optimalizace na zpracování podmínek

- je vidět, že kratší vzory umí CPU zpracovat efektivněji
- zpracování náhodné sekvence tedy zkusíme rozdělit na bloky

Kód:

```
1  const int loops = 32;
2  for( int i = 0; i < loops; i++ )
3     performUnrolledTest( data, size, a );
```

nahradíme kódem

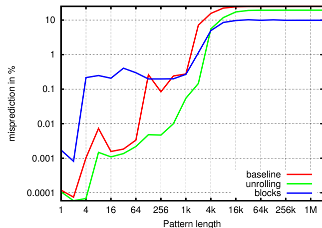
```
1  const int loops = 32;
2  const int blockSize = 32;
3  for( int offset = 0; offset < size; offset += blockSize )
4     for( int i = 0; i < loops; i++ )
5        performUnrolledTest( &data[ offset ], blockSize, a );
```

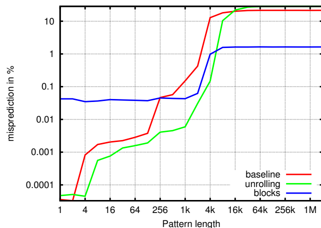
Optimalizace na zpracování podmínek

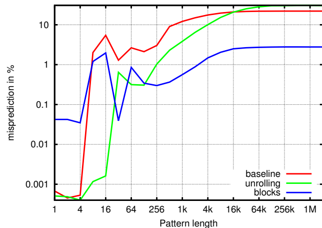
Výsledky (neúspěšně předvídané podmínky):

Délka vzoru	bez rozbalení		s rozbalením		s bloky	
	bez `-O3`	s `-O3`	bez `-O3`	s `-O3`	bez `-O3`	s `-O3`
1	0.0006%	0.0006%	0.0005%	0.0006%	0.02%	0.04%
2	0.0005%	0.0004%	0.0005%	0.0005%	0.02%	0.04%
4	0.0010%	0.0005%	0.0005%	0.0004%	0.02%	0.03%
8	3.9955%	3.0589%	0.0012%	0.0009%	0.94%	1.19%
16	7.1143%	4.6074%	0.0014%	0.0011%	1.16%	1.69%
32	5.1034%	1.4232%	0.0021%	0.0017%	0.02%	0.03%
64	8.0148%	2.2992%	2.0792%	0.3154%	2.66%	0.79%
128	10.9170%	2.5160%	3.9394%	0.0028%	1.74%	0.48%
256	15.0898%	3.4148%	7.3365%	1.0891%	2.04%	0.26%
512	16.9543%	11.1091%	11.9075%	2.3585%	2.24%	0.36%
1024	17.8589%	12.1402%	16.8415%	4.1920%	2.47%	0.56%
2048	18.7427%	14.9048%	20.7197%	6.7165%	2.71%	0.88%
4096	18.6398%	17.6160%	22.6760%	10.5239%	2.87%	1.46%
8192	18.3994%	19.6302%	22.8864%	15.7348%	2.92%	2.04%
16384	18.3008%	21.0191%	22.8314%	20.9794%	2.92%	2.49%
32768	18.3611%	21.6142%	22.8662%	25.6041%	2.95%	2.65%
65536	18.3970%	21.9362%	22.8915%	28.3501%	2.95%	2.72%
131072	18.3810%	22.0045%	22.8386%	29.7677%	2.95%	2.76%
262144	18.3722%	22.0538%	22.8021%	30.1786%	2.94%	2.78%
524288	18.3735%	22.0417%	22.7777%	30.3599%	2.94%	2.78%
1048576	18.3635%	22.0176%	22.7585%	30.3551%	2.93%	2.77%
2097152	18.3620%	22.0096%	22.7403%	30.2972%	2.94%	2.77%

Optimalizace na zpracování podmínek

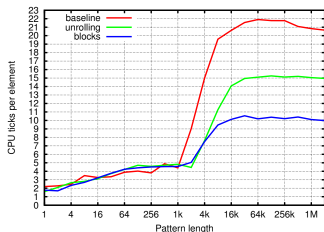


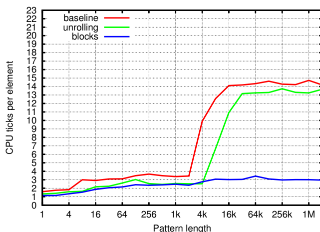


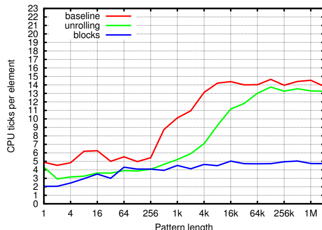


Porovnání efektivity předvídání podmínek na Intel Core 2 E6600, Intel i7 4600M a AMD Phenom 2 X6 1075T s `-O3`.

Optimalizace na zpracování podmínek







Počet taktů CPU na jeden element na Intel Core 2 E6600, Intel i7 4600M a AMD Phenom 2 X6 1075T s `-O3`.

Optimalizace na zpracování podmínek

Můžeme také použít příkaz `switch`:

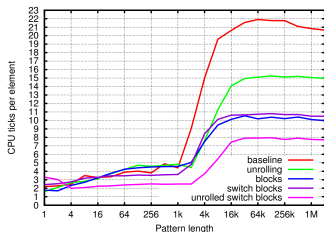
```
1   void performSwitchTest( const int* data,
2                           const int size,
3                           int& a )
4   {
5      a = 0;
6      for( int i = 0; i < size; i ++ )
7      {
8         switch( a[ data ] )
9         {
10           case 1:
11              a++;
12              break;
13           case -1:
14              a--;
15              break;
16           default:
17              a = 0;
18        }
19     }
20  }
```

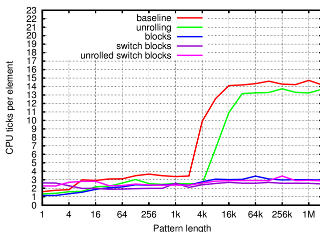
Optimalizace na zpracování podmínek

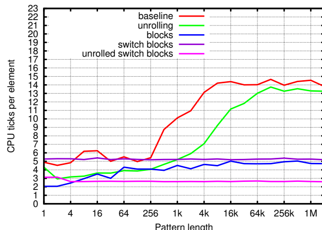
nebo

```
1  void performUnrolledSwitchTest( const int* data,
2                                  const int size,
3                                  int& a )
4  {
5     a = 0;
6     for( int i = 0; i < size; i += 4 )
7     {
8        switch( a[ data ] )
9        {
10          case 1:
11             a++;
12             break;
13          case -1:
14             a--;
15             break;
16          default:
17             a = 0;
18       }
19       switch( a[ data + 1 ] )
20       {
21          case 1:
22             a++;
23             break;
24          case -1:
25             a--;
26             break;
27          default:
28             a = 0;
29       }
30       switch( a[ data + 2 ] )
31       {
32          case 1:
33             a++;
34             break;
35          case -1:
36             a--;
37             break;
38          default:
39             a = 0;
40       }
41       switch( a[ data + 3 ] )
42       {
43          case 1:
44             a++;
45             break;
46          case -1:
47             a--;
48             break;
49          default:
50             a = 0;
51       }
52    }
53 }
```

Optimalizace na zpracování podmínek







Počet taktů CPU na jeden element na Intel Core 2 E6600, Intel i7 4600M a AMD Phenom 2 X6 1075T s `-O3`.

Optimalizace na zpracování podmínek

Nyní přídáme počet podmínek na jeden element pole:

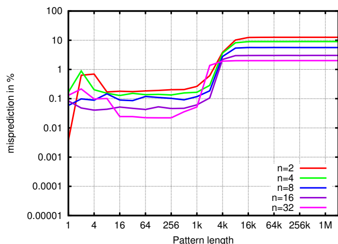
```
 1  void performMultivalueTest( const int* data,
 2                              const int size,
 3                              const int n,
 4                              int& a )
 5  {
 6     a = 0;
 7     for( int i = 0; i < size; i++ )
 8     {
 9        for( int j = -n; j <= n; j++ )
10        {
11           if( data[ i ] == n )
12              a++;
13           if( data[ i ] == -n )
14              a--;
15        }
16        if( data[ i ] == 0 )
17           a = 0;
18     }
19  }
```

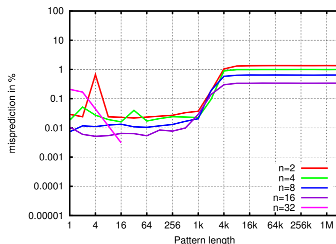
Optimalizace na zpracování podmínek

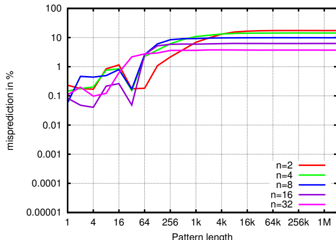
Vnitřní smyčku rozbalíme pomocí C++ šablon:

```
1   template< int n >
2   struct MultivalueTester
3   {
4      static void test( const int data, int& a )
5      {
6         if( data == n )
7            a++;
8         if( data == -n )
9            a--;
10        MultivalueTester< n - 1 >::test( data, a );
11     }
12  };
13
14  template<>
15  struct MultivalueTester< 0 >
16  {
17     static void test( const int data, int& a )
18     {
19        if( data == 0 )
20           a = 0;
21     }
22  };
23
24  template< int n >
25  void performTemplateMultivalueTest( const int* data,
26                                      const int size,
27                                      int& a )
28  {
29     a = 0;
30     for( int i = 0; i < size; i += n )
31     {
32        MultivalueTester< n >::test( data[ i ], a );
33        MultivalueTester< n >::test( data[ i + 1 ], a );
34        MultivalueTester< n >::test( data[ i + 2 ], a );
35        MultivalueTester< n >::test( data[ i + 3 ], a );
36     }
37  }
```

Optimalizace na zpracování podmínek



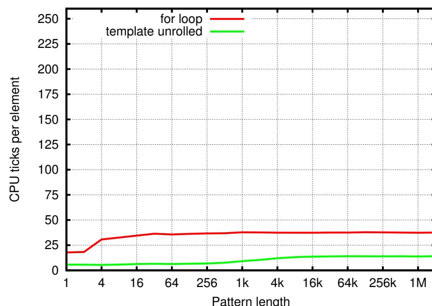


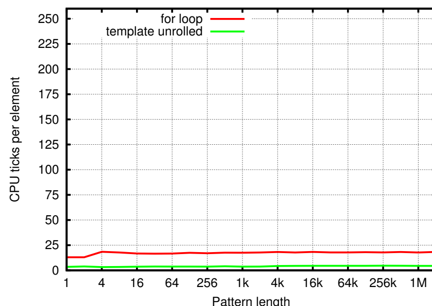


Procento neúspešných předvídání při $2n + 1$ počtu podmínek na element na Intel Core 2 E6600, Intel i7 4600M a AMD Phenom 2 X6 1075T s `-O3`.

Optimalizace na zpracování podmínek

$$\texttt{n = 2}$$

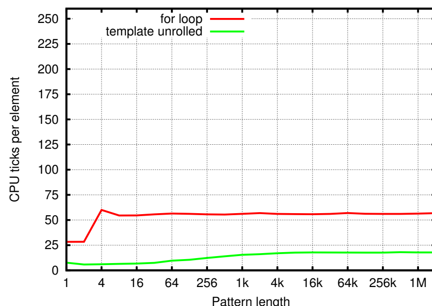


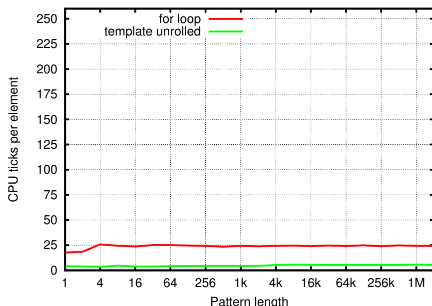


Počet taktů na jeden element na AMD Phenom 2 X6 1075T a Intel i7 4600M s `-O3`.

Optimalizace na zpracování podmínek

$$n = 4$$

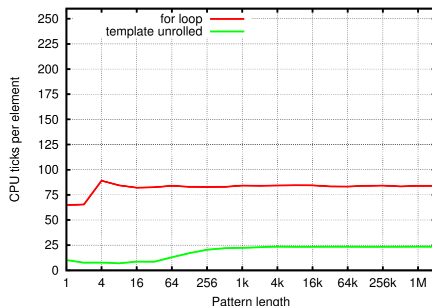


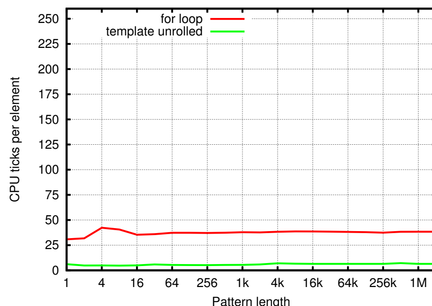


Počet taktů na jeden element na AMD Phenom 2 X6 1075T a Intel i7 4600M s `-O3`.

Optimalizace na zpracování podmínek

`n = 8`

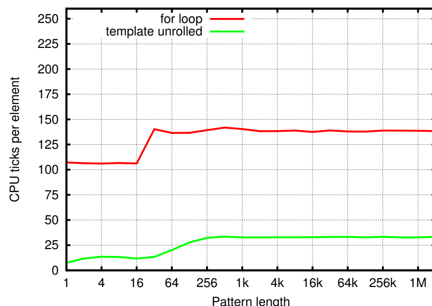


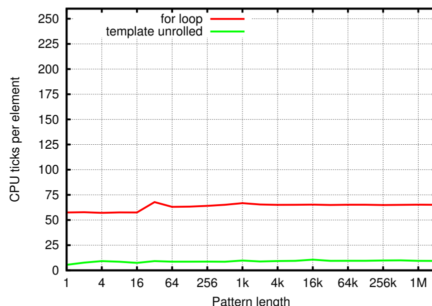


Počet taktů na jeden element na AMD Phenom 2 X6 1075T a Intel i7 4600M s `-O3`.

Optimalizace na zpracování podmínek

`n = 16`

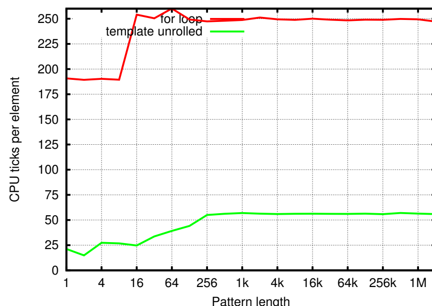


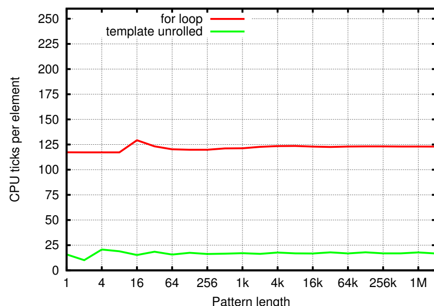


Počet taktů na jeden element na AMD Phenom 2 X6 1075T a Intel i7 4600M s `-O3`.

optimalizace na zpracování podmínek

$$n = 32$$





Počet taktů na jeden element na AMD Phenom 2 X6 1075T a Intel i7 4600M s `-O3`.

optimalizace na zpracování podmínek

- tento příklad celkově ukazuje, že:

Rozbalení krátkého for cyklu pomocí šablon C++ může přinést až 8-násobné urychlení.

Optimalizace na zpracování podmínek

- překladače dokáží provádět rozbalování smyček automaticky
- u `gcc` k tomu slouží přepínač `-funroll-loops`
- podíváme se na jeho efektivitu (`gcc` verze 4.8)

Optimalizace na zpracování podmínek

Rozbalování smyček

Příklad: Výpočet sumy

```
1  for( int i = 0; i < size; i++ )
2      sum += data[ i ];
```

```
1  for( int i = 0; i < size; i+= 2 )
2  {
3      sum += data[ i ];
4      sum += data[ i + 1 ];
5  }
```

```
1  for( int i = 0; i < size; i += 4 )
2  {
3      sum += data[ i ];
4      sum += data[ i + 1 ];
5      sum += data[ i + 2 ];
6      sum += data[ i + 3 ];
7  }
```

...

Optimalizace na zpracování podmínek

Unrolling	Takty na element	
	`-O3`	`-O3 -funroll-loops`
žádný	1.8353	1.14267
2	0.4984	0.44991
4	0.4660	0.46567
8	0.2447	0.23750
16	0.1853	0.13921
32	0.1342	0.12992
64	0.1372	0.12835

Efekt rozbalení smyček na Intel i7 4600M.

Optimalizace na zpracování podmínek

- vidíme, že použití `-funroll-loops` dává lepší výsledky, pokud jsme sami žádný unrolling neprovedli
- pokud provedem unrolling sami, můžeme tím ještě mnoho získat a to až do 16-ti násobného rozbalení
- pokud by tělo for cyklu bylo větší, mohlo by ale dojít k velkému nárůstu kódu a zahlcení instrukční cache
- to, že jeden element zpracujeme jen za jednu osminu taktu CPU je způsobeno vektorizací (viz. další části přednášky)

Volání funkcí

- 64-bitové procesory mají dvojnásobek registrů, lze tedy předávat více parametrů pomocí nich a ne přes zásobník
- lze tak předat až 8 parametrů typu `double/float` a 6 typu `int` nebo ukazatel
- volání virtuální metody obnáší dynamické určení typu a tedy podmíněný skok
- otestujeme si, jak to může snížit výkon

Volání funkcí

Příklad s virtuální metodou:

```
1   class adder
2   {
3      public:
4
5         virtual void addNumber( int& n ) const = 0;
6   };
7
8   class oneAdder : public adder
9   {
10     public:
11
12        void addNumber( int& n ) const { n += data[ 0 ]; data[ 0 ] *= -1; };
13  };
14
15  class twoAdder : public adder
16  {
17     public:
18
19        void addNumber( int& n ) const { n += data[ 1 ]; data[ 1 ] *= -1; };
20  };
```

- pokud bysme do těla metody napsali pouhé `n += 1` nebo `n += 2`, překladač by to eliminoval

Volání funkcí

Příklad s příkazem `switch`:

```
1  class switchAdder
2  {
3     public:
4
5        switchAdder( int c )
6        : number( c ){};
7
8        void addNumber( int& n ) const
9        {
10          switch( this->number )
11          {
12             case 0:
13                n += data[ 0 ];
14                break;
15             case 1:
16                n += data[ 1 ];
17                break;
18             case 2:
19                n += data[ 2 ];
20                break;
21             case 3:
22                n += data[ 3 ];
23                break;
24          }
25          data[ this->number ] *= -1;
26       }
27
28    protected:
29
30       int number;
31 };
```

Volání funkcí

Příklad s šablonovým parametrem:

```
1   template< int index >
2   class templateAdder
3   {
4       public:
5           void addNumber( int& n ) const
6           {
7               n += data[ index ]; data[ index ] *= -1;
8           };
9
10  };
```

- ovšem nahrazení virtuálních metod šablonovými parametry není vždy možné
- a pokud ano, vyžaduje často výraznou změnu návrhu aplikace

Volání funkcí

Výsledky testu:

	Intel Core2	Intel i7	AMD Phenom 2
Virtuální metoda	7.08	5.85	7.11
`switch`	1.77	0.83	1.78
Nevirtuální metoda	1.90	0.84	1.89
Šablonový parametr	1.96	0.89	1.91
`inline` metoda	1.89	0.81	1.88
Bez volání funkce	1.82	0.81	1.82

- vidíme, že volání virtuální metody je cca. 4-krát pomalejší
- ostatní volání vychází stejně
- volání pomocí `switch` je rychlejší, CPU zřejmě nedokáže předvídat volání virtuálních metod
- implementace bez volání funkce a `inline` volání vychází stejně
- šlo o velmi jednoduché fce., takže je překladač zřejmě zpracoval jako `inline`
- pokud se všechny parametry vejdou do registrů, nemusí nás volání funkce stát nic

Eliminace podmínek

- standard C/C++ definuje, že je-li např. ve výrazu `podmínkaA && podmínkaB` podmínkaA nesplněna, podmínkaB se už nevyhodnocuje
- proto je lepší zpisovat dříve podmínky, které budou pravděpodobně nesplněny nebo podmínky, které se snadno vyhodnotí
- podobná úvaha platí i pro operaci `||`

Eliminace podmínek

- často používanou podmínku

```
1   if( i >= 0 && i < size)
```

nahradit podmínkou

```
1   if( ( unsigned int ) i < size )
```

- přetypovaní na `unsigned int` nestojí vůbec nic a je-li `i` zaporné celé číslo, stane se z něj po přetypování hodně velké kladné číslo, takže podmínce nevyhoví
- podobně podmínku

```
1   if( i >= c && i < size)
```

lze nahradit za

```
1   if( ( unsigned int ) i - c < size - c )
```

kde výraz `size - c` lze předpočítat

Zpracování algebraických výrazů

- výraz `a + b + c + d` je ekvivalentní výrazu `( a + b ) + ( c + d )`
- první je sekvence tří operací
- druhý je suma dvou výrazů, které mohou být zpracovány nezávisle
- CPU tak může využít superskalární zpracování kódu a lépe využít pipeline
- překladače tyto úpravy provádějí automaticky
- v připadě aritmetiky s plovoucí desetinou čárkou ale oba výrazy nejsou ekvivalentní
- překladač by je tedy neměl zaměňovat, pokud nepoužíjeme přepínač `-ffast-math`

Zpracování algebraických výrazů

Příklad: Porovnáme následující tři zápisy stejného výrazu.

```
1  sum += n1 + n2 + n3 + n4 + n5 + n6 + n7 + n8;
2  sum += ( n1 + n2 ) + ( n3 + n4 ) + ( n5 + n6 ) + ( n7 + n8 );
3  sum += ( ( n1 + n2 ) + ( n3 + n4 ) ) + ( ( n5 + n6 ) + ( n7 + n8 ) );
```

Výsledek:

Výraz	Intel i7 4600M			AMD Phenom 2 X6 1075T		
	`int`	`float`	`double`	`int`	`float`	`double`
1	1.6	1.6	3.2	2.0	3.0	5.8
2	1.4	1.5	3.0	1.9	2.8	5.4
3	1.4	1.5	3.0	1.9	2.6	5.0

Počet taktů na zpracování daných výrazů s překladačem gcc-4.8.2 s `-O3`.

- výsledky meření to ale nepotvrzují a ukazuje se, že překladač optimalizuje všechny tři výrazy téměř stejně
- u úloh s velkou citlivostí na přesnost výpočtu je tedy dobré dávat pozor na optimalizace překladače

Zpracování algebraických výrazů

Náročnost některých celočíselných operací s typem `int`.

Operace	Intel i7 4600M	AMD Phenom 2 X6
`*`	0.5	0.6
`/`	2.2	3.6
`% 2`	0.6	0.8
`% 3`	2.3	4.3

- ve výsledcích je započítána možnost vektorizace
- `% 2` překladač optimalizuje pomocí bitového OR

Zpracování algebraických výrazů

Náročnost některých operací s plovoucí desetinnou čárkou.

Operace	Intel i7 4600M		AMD Phenom 2 X6	
	`float`	`double`	`float`	`double`
`*`	0.6	1.2	0.8	1.9
`/`	1.5	5.7	3.4	7.6
`pow( x, 2.0)`	19.1	1.8	8.9	1.2
`pow( x, 2.13)`	150.3	146.6	229	229
`sqrt`	16	16	34	23
`exp`	49	46	96	94
`log`	73	71	149	147
`sin`	58	53	119	118
`int ->`	0.7	1.2	1.5	3.9
`float ->`	–	1.3	–	4.1

- konstanty jako `1.0/3.0` jsou typu `double`, pří výpočtech s typem `float` je třeba psát `( float ) 1.0/3.0` jinak dojde k výpočtu v typu `double`

Výjimky v C++

- výjimky v C++ jsou efektivní nástroj pro ošetření výjimečných situací
- při vzniku výjimky je potřeba umět ji propagovat nahoru zásobníkem
- zároveň se volají všechny potřebné destruktory
- to je náročný proces a je potřeba se na něj připravit
- dnešní překladače implementují tzv. *zero-cost exception handling*
- to znamená, že pokud výjimka nevznikne, nemělo by použití výjimek v kódu naši aplikaci zpomalit
- ošetření vzniklé výjimky je s tímto přístupem pomalější, ale to už nevadí

Výjimky v C++

Příklad: Efektivta zpracování kódu s výjimkami.

- porovnáme následující dvě funkce

```
1   void performTest( const int* data,
2                     const int size,
3                     int& a )
4   {
5      a = 0;
6      for( int i = 0; i < size; i++ )
7      {
8         if( data[ i ] == 1 )
9            a++;
10        if( data[ i ] == 0 )
11           a = 0;
12     }
13  }
```

```
1   void performTestWithException( const int* data,
2                                  const int size,
3                                  int& a )
4   {
5      a = 0;
6      for( int i = 0; i < size; i++ )
7      {
8         if( data[ i ] == 1 )
9            a++;
10        if( data[ i ] == 0 )
11           throw -1;
12     }
13  }
```

Výjimky v C++

Výsledky:

	AMD Phenom 2 X6 1075T	Intel i7 4600M
bez výjimky	2.1	1.28
s výjimkou	2.7	1.29
zpracování výjimky	6408	4179

- tabulka udává výsledný počet taktů CPU na zpracování jednoho elementu, pokud nevznikne žádná výjimka
- poslední řádek říká, kolik taktů zabere zpracování vzniklé výjimky

Výjimky v C++

- vidíme, že *zero-cost* zpracování výjimek je opravdu efektivní, obzvlášť na novějších CPU
- použití výjimek navíc může eliminovat podmínky pro testování chybových kódů, což může výpočet dokonce ještě trochu urychlit

Aserce

- ošetření výjimečných stavů, ke kterým by nemělo dojít, pokud program dobře funguje, je také možné pomocí asercí a makra `assert`
- program se musí dobře otestovat a v produkční verzi se aserce úplně vyřadí

Disassembling

- při optimalizování kódu se často hodí mít možnost nahlédnout do výsledného strojového kódu vytvořeného překladačem
- tomu se říká disassembling
- šikovně lze využít debuger `gdb`
- náš program spustíme příkazem
  - `gdb -args program argumenty`
- v gdb zadáme příkaz
  - `disas` *jméno funkce*
- samozřejmě se je lepší mít kód přeložený s volbou `-g`

Shrnutí

- rychlost zpracování instrukcí v CPU je výrazně ovlivněna efektivitou využití pipeline
- tu nejvíc narušují podmíněné skoky
- CPU zpracovává kód spekulativně a za určitých okolností je schopný podmíěné skoky dobře odhantou dopředu
- vyplatí se podmínky eliminovat, zpracovávat data po menších blocích a provádět *loop unrolling*
- zpracování výjimek v C++ lze dnes považovat za bezproblémové z pohledu rychlosti zpracování kódu