W2 Zrównoleglenie i przetwarzanie potokowe

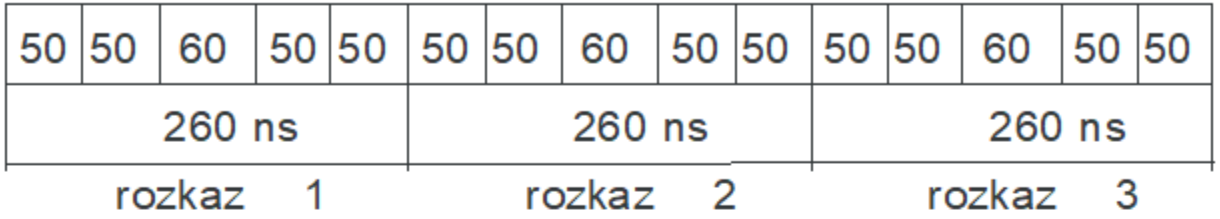
- **Zrównoleglenie – wysoka wydajność pozostaje osiągnięta w efekcie jednoczesnego wykonania różnych części zagadnienia.**

- **Przetwarzanie potokowe – proces przetwarzania jest rozdzielony na drobne części – stopni przebiegu. Każdy stopień będzie wykonany oddzielnym blokiem sprzętu. Przetwarzanie dowolnego rozkazu można rozdzielić na kilka stopni i zorganizować przekazanie danych od jednego etapu do innego. Przetwarzanie potokowe można wykorzystać dla skojarzenia etapów wykonania różnych rozkazów. Wydajność rośnie w efekcie tego, że na różnych stopniach przetwarzania potokowego jednocześnie są wykonane kilka rozkazów.**

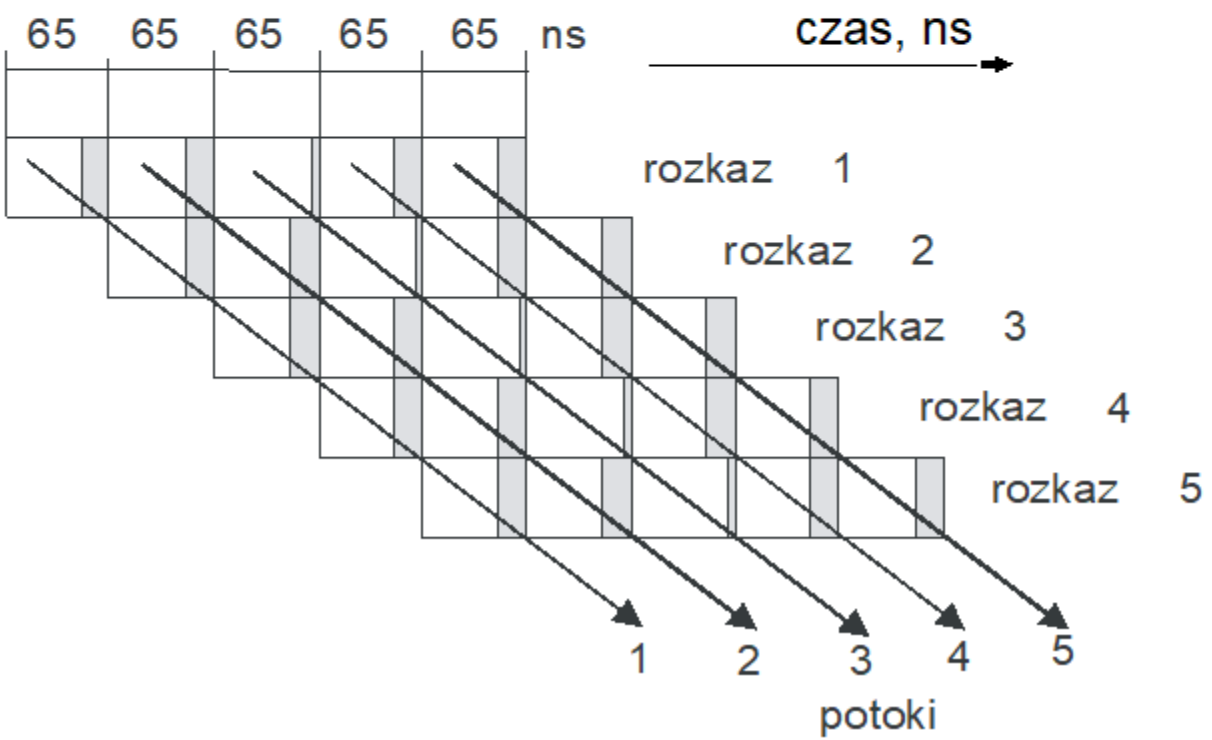
Przykład: mamy 5 etapów wykonania operacji trwałością 50, 50, 60, 50 i 50 ns odpowiednio, a 5 ns – nakład na organizacje przetwarzania potokowego. Średni czas wykonania rozkazu przy przetwarzani sekwencyjnym wynosi 260 ns, a przy przetwarzani potokowym (t_{sr}^{pot}) – trwałości najdłuższego taktu plus nakład na przetwarzanie potokowe – 60+5 = 65 ns [t_{sr}^{pot} = (maksymalna trwałość etapu)*(ilość etapów)/(ilość potoków) = 65*5/5 = 65 ns]. Więc przyspieszenie wynosi 260/65 = 4 razy.

Większość współczesnych procesorów używają przetwarzanie potokowe.

Sekwencyjne przetwarzanie rozkazów



Potokowe przetwarzanie rozkazów



W2

Typowe etapy przetwarzania typowego rozkazu:

IF - obliczenie adresu rozkazu, pobranie rozkazu
ID - dekodowanie rozkazu
OF - obliczenie adresu operandu, pobranie operandu
EX - wykonanie operacji na operandach
WB - zapisanie wyniku

Hazardy danych:

Przykład 1

```
for(i=0; i<N; i++)
{
        A[i] = 1.1* A[i];
}
```

Potok, pobierający wartość operandu A[…] (etap OF) na iteracji i+1 , będzie wstrzymany dopóki licznik iteracji pętli *i* nie zostanie inkrementowany.

Rozwiązanie:

```
for(i=0; i<N; i+=2)
{
        A[i]   =  1.1* A[i];
        A[i+1] =  1.1*A[i+1];
}
```

Instrukcji w pętli są niezależne → potoki procesora w stanie działać równoległe.

Czas ładowania w rejestr N zależnych słów wynosi

$T = N \cdot (T_{ch} + T_{mem})$, gdzie T_{ch} – latentność chipseta, T_{mem} – latentność pamięci.

Czas ładowania N niezależnych słów wynosi

$T = N/C + T_{ch} + T_{mem}$, gdzie C – przepustowość układu pamięci.

(Definicja: C = N/t, gdzie t – odcinek czasu, za który będą przesłane N słów. Z tego wynika: t = N/C)

Przykład 2

```
for(i=0; i<N; i++)
{
    A[i] = A[i]+B[i];    //line 1
    C[i] = A[i]+1.0;     //line 2
    A[i] = D[i]+1.0;     //line 3
}
```

Linia kodu 3 nie może być wykonana dopóki nie będzie zakończona linia 2. Inaczej A[i] będzie zmodyfikowano wcześniej od jego użycia w linii 2.

Rozwiązanie:

```
register double rrr;
for(i=0; i<N; i++)
{
    A[i] = A[i]+B[i];    //line 1
    rrr = A[i];
    C[i] = rrr+1.0;      //line 2
    A[i] = D[i]+1.0;     //line 3
}
```

Wpływ różnych technik programowania na szybkość wykonania algorytmu.

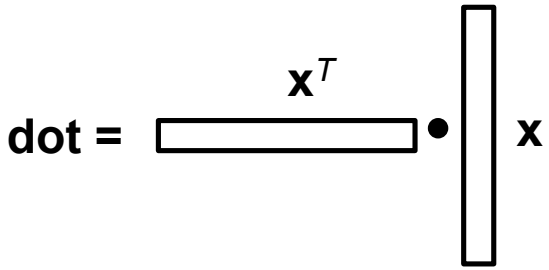
Przykład MemTest_1:

Obliczenie iloczynu skalarnego $dot = x^T * x$ **przy zastosowaniu różnych technik.**

Charakterystyki komputera:

- **Procesor – Intel® Core™2 Quad CPU Q6600 @2.40 GHz**
- **Cache - L1: 32 KB, L2:4096 KB**
- **Pamięć: DDR2 800 MHz 4 GB**

Technika SSE2: dot = $x^T * x$



$$dot = \sum_{k=1}^{n} x_k^2$$

load_pd: c1 = $\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$ c2 = $\begin{bmatrix} x_3 \\ x_4 \end{bmatrix}$ c3 = $\begin{bmatrix} x_5 \\ x_6 \end{bmatrix}$ c4 = $\begin{bmatrix} x_7 \\ x_8 \end{bmatrix}$

mul_pd: c1 = $\begin{bmatrix} x_1^2 \\ x_2^2 \end{bmatrix}$ c2 = $\begin{bmatrix} x_3^2 \\ x_4^2 \end{bmatrix}$ c3 = $\begin{bmatrix} x_5^2 \\ x_6^2 \end{bmatrix}$ c4 = $\begin{bmatrix} x_7^2 \\ x_8^2 \end{bmatrix}$

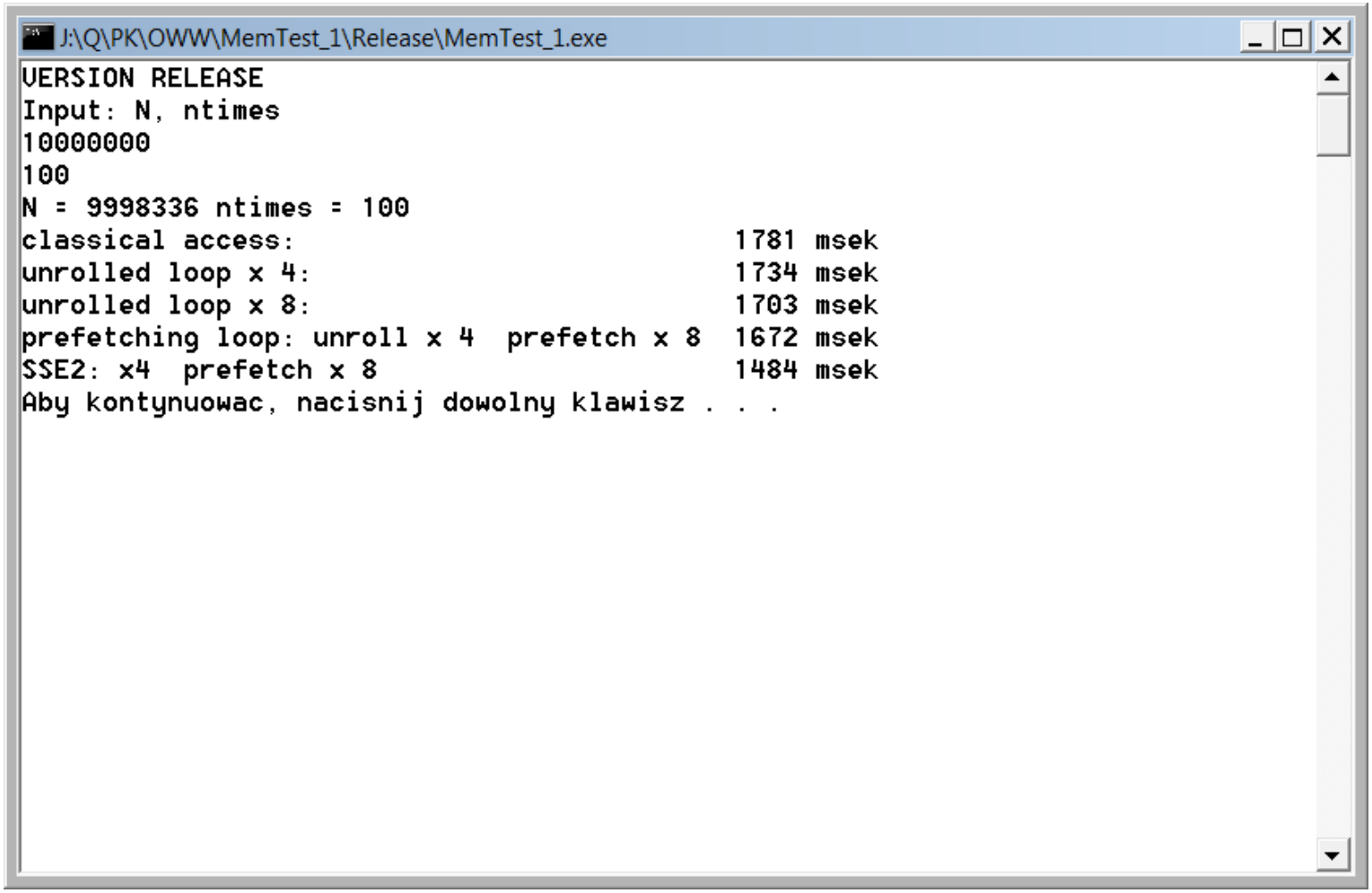
sum = $\begin{bmatrix} x_1^2 + x_3^2 + \ldots + x_{2n-1}^2 \\ x_2^2 + x_4^2 + \ldots + x_{2n}^2 \end{bmatrix}$

$$dot = sum[0] + sum[1];$$

Technika SSE2: dot = X^T*X

```
sum = _mm_setzero_pd();

for(i=0; i<N; i+=8)
{
    _mm_prefetch((const char *)(&X[i+8]), _MM_HINT_T0);
    c1 = _mm_load_pd(&X[i]);     //load X[i], X[i+1]   to c1
    c2 = _mm_load_pd(&X[i+2]);   //load X[i+2], X[i+3] to c2
    c3 = _mm_load_pd(&X[i+4]);   //load X[i+4], X[i+5] to c3
    c4 = _mm_load_pd(&X[i+6]);   //load X[i+6], X[i+7] to c4
    c1 = _mm_mul_pd(c1, c1);    //tmp <- c1*c1
    sum = _mm_add_pd(sum, c1);  //sum = sum + c1*c1
    c2 = _mm_mul_pd(c2, c2);    //tmp <- c2*c2
    sum = _mm_add_pd(sum, c2);  //sum = sum + c1*c1 + c2*c2
    c3 = _mm_mul_pd(c3, c3);    //tmp <- c3*c3
    sum = _mm_add_pd(sum, c3);  //sum = sum + c1*c1 + c2*c2 + c3*c3
    c4 = _mm_mul_pd(c4, c4);    //tmp <- c4*c4
    sum = _mm_add_pd(sum, c4);  //sum = sum + c1*c1 + c2*c2 + c3*c3 + c4*c4
}
_mm_store_pd (res, sum);         //unload res <- sum
dot1 = res[0]+res[1];            //finally: dot1 = res[0]+res[1];
```



Parametry rozwijania pętli i prefetchu istotnie zależą od architektury procesora, częstości magistrali i wielu różnych czynników.

Charakterystyki komputera:

Procesor – Intel® Core™ i7 2760QM clock rate 2400 - 3400 MHz

Cache - L1: 32 KB, L2:256 KB, L3: 6144 KB

Pamięć: DDR3 665.2 MHz 8 GB

```
VERSION RELEASE
Input: N, ntimes
10000000
100
N = 9998336 ntimes = 100
classical access:                                1138 msek
unrolled loop x 4:                                749 msek
unrolled loop x 8:                                702 msek
prefetching loop: unroll x 8  prefetch x 8        733 msek
SSE2: x4  prefetch x 8                            671 msek
Aby kontynuowac, nacisnij dowolny klawisz . . .
```

Prefetch – pobieranie danych z wyprzedzeniem z celą ukryć zwłokę (latentność) układu pamięci.
Typowy schemat bez prefetch'u:

```
for(i=0; i<N; i++)
        dot += X[i]*X[i];
```

i=0; procesor wystawia na chipset zapotrzebowanie na słowo X[0]. Chipset stawi to w kolejkę zapotrzebowani. Jak tylko opracowanie tej kolejki dojdzie do naszego zapotrzebowania, podane słowo będzie odczytane z pamięci głównej i za pomocą magistrali przesunięte do cache L3. Procesor w tym czasie wykonuje puste cykle. Odcinek czasu licząc od momentu wystawienia zapotrzebowania na chipset do umieszczenia danych w cache jest latentnością układu pamięć – chipset ($T_{ch}+T_{mem}$).

i=1; Wszystko będzie powtórzone dokładnie tak, jak dla poprzedniej iteracji.

……

Czas wykonania takiego algorytmu : $T = N\cdot(T_{ch}+T_{mem}) + 2\cdot N\cdot t_{ops} \approx$
$\approx N\cdot(T_{ch}+T_{mem})$, gdzie t_{ops} – czas wykonania jednego mnożenia lub dodawania.

Typowy schemat z prefetch'em:

```
for(i=0; i<N; i+=8)
{
        //inicjujemy ładowanie następnej linii cache danych za adresem
        //&X[i+8] do cache L1. Przy niemożliwości ładowania cache L1
        //będzie ładowany cache L2. Zakładamy, że rozmiar cache linii
        //dla podanego sprzętu wynosi 64 B = 8 słów double. Ten kod jest
        //zależny od sprzętu.

        _mm_prefetch((const char *)(&X[i+8]), _MM_HINT_T0);
        dot += X[i]*X[i]+X[i+1]*X[i+1] +…+X[i+7]*X[i+7];
}
```

Teraz oczekiwanie procesora w skutek latentności układu pamięci odbywa się tylko przy i = 0. Dalej procesor i układ pamięci działają równoległe. Przyczyną hamowania obliczeń dla takiego kodu jest ograniczona przepustowość magistrali, a nie latentność układu pamięci.

Czas wykonania takiego algorytmu: $T = N/C + T_{ch}+T_{mem} + 2\cdot N\cdot t_{ops}/k \approx N/C$, gdzie C – przepustowość układu pamięci, k – przyspieszenie obliczeń w skutek usunięcia zaburzeń potokowego przetwarzania danych.

http://www.docstoc.com/docs/284306/MMCodec_amd64

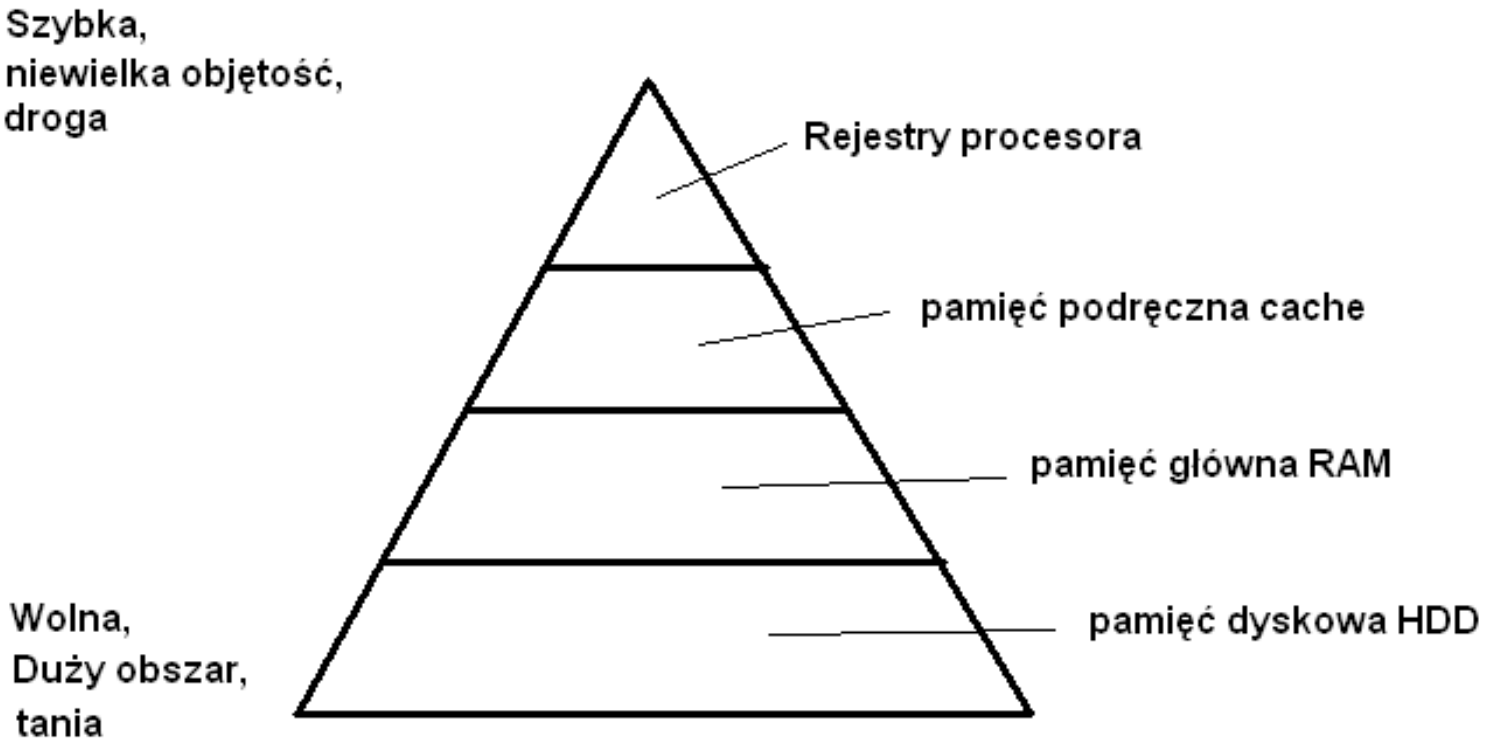
The hardware prefetcher brings in 64 bytes (cache line) at a time. When the hardware prefetcher detects an access to cache line L, followed by cache line L+1, it will initiate a prefetch to the subsequent cache line.

The hardware prefetcher has some limitations. It cannot cross page boundaries. It only fills the L2 cache. It may not detect access patterns with large strides. All these limitations can be overcome with the correct use of software prefetches.

Software prefetches can also be given hints about different kinds of data. For data that is read and will be needed again soon, developers should use the "Read" prefetches. For data that is discarded after the first use, developers should use the "Non-temporal read" version of prefetch. For data, which will soon be written to, developers should use the "Write" prefetches.

SSE2 Intrinsics	SSE2 Instructions	Type of prefetch	Cache filled and Cache evicted to.
_mm_prefetch(void* address,_MM_HINT_T0)	prefetch	Read prefetch	Fills L1 data cache. When evicted, data sent to L2 data cache.
_mm_prefetch(void* address,_MM_HINT_NTA)	prefetchnta	Non-temporal read prefetch	Fills one way of L1 data cache. When evicted, data sent directly to memory if modified and not to L2 data cache.
_m_prefetchw(void* address)	prefetchw	Write prefetch.	Fills L1 data cache. When evicted, data sent to L2 data cache.

Hierarchia pamięci



Przykładowe rozmiary i czasy dostępu dla różnych rodzajów pamięci:

Typ pamięci	rozmiar	Czas dostępu, ns
rejestry	Kilkaset B	< 1 ns
podręczna (L1)	kilkanaście KB	Kilka ns
podręczna (L2 - L3)	Kilka MB	Kilkanaście ns
główna	Kilkaset MB	sto kilkadziesiąt ns

Optymalne wykorzystanie pamięci podręcznej jest bardzo ważne dla wydajności programów