Pamięci podręczne

October 28, 2014

Problem: Procesory są tanie i szybkie.

Problem: Procesory są tanie i szybkie.
Pamięci są tanie, szybkie, pojemne

Problem: Procesory są tanie i szybkie.
Pamięci są tanie, szybkie, pojemne (wybierz dwa z trzech).

Problem: Procesory są tanie i szybkie.
Pamięci są tanie, szybkie, pojemne (wybierz dwa z trzech).
Szybki procesor z wolną pamięcią nie ma sensu.
(chyba, że chcemy rozwiązywać jedynie problemy o złożoności pamięciowej $O(1)$)

Historia: Kiedyś procesory były drogie i wolne.

Problem: Procesory są tanie i szybkie.
Pamięci są tanie, szybkie, pojemne (wybierz dwa z trzech).
Szybki procesor z wolną pamięcią nie ma sensu.
(chyba, że chcemy rozwiązywać jedynie problemy o złożoności pamięciowej $O(1)$)

Historia: Kiedyś procesory były drogie i wolne.
Ale pamięci były jeszcze droższe i dużo wolniejsze.

Problem: Procesory są tanie i szybkie.
Pamięci są tanie, szybkie, pojemne (wybierz dwa z trzech).
Szybki procesor z wolną pamięcią nie ma sensu.
(chyba, że chcemy rozwiązywać jedynie problemy o złożoności pamięciowej $O(1)$)

Historia: Kiedyś procesory były drogie i wolne.
Ale pamięci były jeszcze droższe i dużo wolniejsze.
Pierwsze cache już w latach 60tych (mainframe'y).

Problem: Procesory są tanie i szybkie.
Pamięci są tanie, szybkie, pojemne (wybierz dwa z trzech).
Szybki procesor z wolną pamięcią nie ma sensu.
(chyba, że chcemy rozwiązywać jedynie problemy o złożoności pamięciowej $O(1)$)

Historia: Kiedyś procesory były drogie i wolne.
Ale pamięci były jeszcze droższe i dużo wolniejsze.
Pierwsze cache już w latach 60tych (mainframe'y).
W komputerach PC dopiero, gdy 386 rozpędził się do 20MHz.

Idea: Umieśćmy szybką pamięć pomiędzy CPU a główną pamięcią.

Idea: Umieśćmy szybką pamięć pomiędzy CPU a główną pamięcią. Szybką – więc małą (bo nie chcemy wydać fortuny).

Idea: Umieśćmy szybką pamięć pomiędzy CPU a główną pamięcią. Szybką – więc małą (bo nie chcemy wydać fortuny).

Pamięć RAM w komputerze – technologia DRAM.
(naładowane / rozładowane kondensatory)
Pamięć cache – technologia SRAM.
(przerzutniki, czyli bramki logiczne)

Procesor pobierając dane patrzy do cache; jeśli znajdzie, pobiera je stamtąd i działa dalej; jeśli nie znajdzie, szuka w pamięci głównej.

Idea: Umieśćmy szybką pamięć pomiędzy CPU a główną pamięcią. Szybką – więc małą (bo nie chcemy wydać fortuny).

Pamięć RAM w komputerze – technologia DRAM.
(naładowane / rozładowane kondensatory)
Pamięć cache – technologia SRAM.
(przerzutniki, czyli bramki logiczne)

Procesor pobierając dane patrzy do cache; jeśli znajdzie, pobiera je stamtąd i działa dalej; jeśli nie znajdzie, szuka w pamięci głównej.

Miary: hit rate (jak często trafiamy)

Idea: Umieśćmy szybką pamięć pomiędzy CPU a główną pamięcią. Szybką – więc małą (bo nie chcemy wydać fortuny).

Pamięć RAM w komputerze – technologia DRAM.
(naładowane / rozładowane kondensatory)
Pamięć cache – technologia SRAM.
(przerzutniki, czyli bramki logiczne)

Procesor pobierając dane patrzy do cache; jeśli znajdzie, pobiera je stamtąd i działa dalej; jeśli nie znajdzie, szuka w pamięci głównej.

Miary: hit rate (jak często trafiamy), hit time (jak szybko potrafimy pobrać dane z cache)

Idea: Umieśćmy szybką pamięć pomiędzy CPU a główną pamięcią. Szybką – więc małą (bo nie chcemy wydać fortuny).

Pamięć RAM w komputerze – technologia DRAM.
(naładowane / rozładowane kondensatory)
Pamięć cache – technologia SRAM.
(przerzutniki, czyli bramki logiczne)

Procesor pobierając dane patrzy do cache; jeśli znajdzie, pobiera je stamtąd i działa dalej; jeśli nie znajdzie, szuka w pamięci głównej.

Miary: hit rate (jak często trafiamy), hit time (jak szybko potrafimy pobrać dane z cache), miss penalty (ile kosztuje ściągnięcie danych z głównej pamięci).

filmik (YAY!)

Problem: Jak zorganizować pamięć cache? Które fragmenty głównej pamięci w niej trzymać?

Problem: Jak zorganizować pamięć cache? Które fragmenty głównej pamięci w niej trzymać?

Prosto: *direct-mapping*;
Uniwersalnie: *full-associative*;
Praktycznie: rozwiązanie pośrednie.

Problem: Jak zorganizować pamięć cache? Które fragmenty głównej pamięci w niej trzymać?

Prosto: *direct-mapping*;
Uniwersalnie: *full-associative*;
Praktycznie: rozwiązanie pośrednie.

Kluczowy czynnik: „geometria" pamięci cache (jak duża, jak zorganizowana).

Problem: Jak zorganizować pamięć cache? Które fragmenty głównej pamięci w niej trzymać?

Prosto: *direct-mapping*;
Uniwersalnie: *full-associative*;
Praktycznie: rozwiązanie pośrednie.

Kluczowy czynnik: „geometria" pamięci cache (jak duża, jak zorganizowana). Niestety – im „fajniejsza" pamięć, tym wolniejsza.

Problem: Jak zorganizować pamięć cache? Które fragmenty głównej pamięci w niej trzymać?

Prosto: *direct-mapping*;
Uniwersalnie: *full-associative*;
Praktycznie: rozwiązanie pośrednie.

Kluczowy czynnik: „geometria" pamięci cache (jak duża, jak zorganizowana). Niestety – im „fajniejsza" pamięć, tym wolniejsza.

Pomysł: cache-cepcja, czyli wiele poziomów cache'a:
pierwszy (L1) – mały, prosty i szybki;
drugi (L2) – większy, bardziej złożony i wolniejszy;
trzeci, czwarty – używany zależnie od humoru projektanta.

Strategie zapisu: Odczyt jest prosty: jeżeli masz dane w cache, użyj ich; jeśli nie, ściągnij je z RAMu do cache i użyj ich.

Strategie zapisu: Odczyt jest prosty: jeżeli masz dane w cache, użyj ich; jeśli nie, ściągnij je z RAMu do cache i użyj ich.

Dwa pomysły na zapis:
write-through: zapis ląduje w cache i jednocześnie jest inicjowany (w tle) zapis do pamięci głównej;
write-behind: zapis ląduje w cache i zostanie przekazany do pamięci głównej, gdy blok cache'a będzie wymieniany.

Strategie zapisu: Odczyt jest prosty: jeżeli masz dane w cache, użyj ich; jeśli nie, ściągnij je z RAMu do cache i użyj ich.

Dwa pomysły na zapis:
write-through: zapis ląduje w cache i jednocześnie jest inicjowany (w tle) zapis do pamięci głównej;
write-behind: zapis ląduje w cache i zostanie przekazany do pamięci głównej, gdy blok cache'a będzie wymieniany.

Zalety, wady, *write-buffer*, semantyka zapisów, urządzenia mapowane w pamięci, *write-combining*, framebuffery, *MTRR*.

W tym miejscu kończy się powtórzenie tego, co było na ASK.

Obwserwacja: W niektórych programach *full-associative* cache nie przynosi wielkich korzyści.

Obwserwacja: W niektórych programach *full-associative* cache nie przynosi wielkich korzyści. W innych stanowi rewolucję. Niestety pamięć L1 nie może mieć zbyt dużej *associativity*, ponieważ musi być szybka.

Obwserwacja: W niektórych programach *full-associative* cache nie przynosi wielkich korzyści. W innych stanowi rewolucję. Niestety pamięć L1 nie może mieć zbyt dużej *associativity*, ponieważ musi być szybka.

Pomysł:

1. Skonstruuj małą *full-associative* pamięć „L1.5".
2. Umieszczaj w niej bloki, które wyleciały z L1.
3. Nazwij ją *victim-cache*.
4. ... (?)
5. PROFIT!

Obwserwacja: W niektórych programach *full-associative* cache nie przynosi wielkich korzyści. W innych stanowi rewolucję. Niestety pamięć L1 nie może mieć zbyt dużej *associativity*, ponieważ musi być szybka.

Pomysł:

1. Skonstruuj małą *full-associative* pamięć „L1.5".
2. Umieszczaj w niej bloki, które wyleciały z L1.
3. Nazwij ją *victim-cache*.
4. ... (?)
5. PROFIT!

Miss w pamięci L1 szuka równolegle w małym, ale za to *full-associative victim-cache* jak i w dużym L2. Programy, w których *full-associativity* pomaga zyskują, pozostałe programy nie tracą.

Obserwacja: Potrzebujemy dostępu do pamięci w wielu miejscach potoku przetwarzania: dekodowanie instrukcji, mapowanie adresów wirtualnych, pobieranie / zapisywanie danych.

Obserwacja: Potrzebujemy dostępu do pamięci w wielu miejscach potoku przetwarzania: dekodowanie instrukcji, mapowanie adresów wirtualnych, pobieranie / zapisywanie danych.

Pomysł: Więcej cache-y! Osobny cache L1 dla instrukcji, osobny dla danych (zwane *icache* oraz *dcache*), osobny dla tablic stron (zwany *TLB* – translation lookaside buffer).

Obserwacja: Potrzebujemy dostępu do pamięci w wielu miejscach potoku przetwarzania: dekodowanie instrukcji, mapowanie adresów wirtualnych, pobieranie / zapisywanie danych.

Pomysł: Więcej cache-y! Osobny cache L1 dla instrukcji, osobny dla danych (zwane *icache* oraz *dcache*), osobny dla tablic stron (zwany *TLB* – translation lookaside buffer). Albo lepiej – osobne *TLB* dla danych, osobne dla instrukcji (w końcu używamy go w innych miejscach potoku).

Obserwacja: Potrzebujemy dostępu do pamięci w wielu miejscach potoku przetwarzania: dekodowanie instrukcji, mapowanie adresów wirtualnych, pobieranie / zapisywanie danych.

Pomysł: Więcej cache-y! Osobny cache L1 dla instrukcji, osobny dla danych (zwane *icache* oraz *dcache*), osobny dla tablic stron (zwany *TLB* – translation lookaside buffer). Albo lepiej – osobne *TLB* dla danych, osobne dla instrukcji (w końcu używamy go w innych miejscach potoku).

Zyski: możemy skonstruować każdy cache inaczej, stosownie do rodzaju danych, które trzyma.

Obserwacja: Potrzebujemy dostępu do pamięci w wielu miejscach potoku przetwarzania: dekodowanie instrukcji, mapowanie adresów wirtualnych, pobieranie / zapisywanie danych.

Pomysł: Więcej cache-y! Osobny cache L1 dla instrukcji, osobny dla danych (zwane *icache* oraz *dcache*), osobny dla tablic stron (zwany *TLB* – translation lookaside buffer). Albo lepiej – osobne *TLB* dla danych, osobne dla instrukcji (w końcu używamy go w innych miejscach potoku).

Zyski: możemy skonstruować każdy cache inaczej, stosownie do rodzaju danych, które trzyma.
TLB przechowuje zazwyczaj kilkadziesiąt-kilkaset wpisów; może być mały i *full-associative*.

Obserwacja: Potrzebujemy dostępu do pamięci w wielu miejscach potoku przetwarzania: dekodowanie instrukcji, mapowanie adresów wirtualnych, pobieranie / zapisywanie danych.

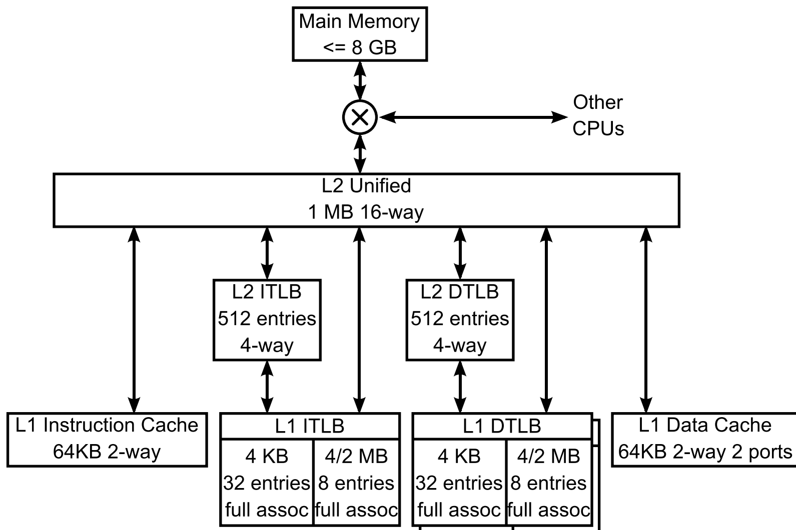
Pomysł: Więcej cache-y! Osobny cache L1 dla instrukcji, osobny dla danych (zwane *icache* oraz *dcache*), osobny dla tablic stron (zwany *TLB* – translation lookaside buffer). Albo lepiej – osobne *TLB* dla danych, osobne dla instrukcji (w końcu używamy go w innych miejscach potoku).

Zyski: możemy skonstruować każdy cache inaczej, stosownie do rodzaju danych, które trzyma.
TLB przechowuje zazwyczaj kilkadziesiąt-kilkaset wpisów; może być mały i *full-associative*.
Cache danych może mieć dwa porty i obsługiwać dwa zapytania jednocześnie (np. jeżeli mamy dwa potoki wykonania).

Obserwacja: Potrzebujemy dostępu do pamięci w wielu miejscach potoku przetwarzania: dekodowanie instrukcji, mapowanie adresów wirtualnych, pobieranie / zapisywanie danych.

Pomysł: Więcej cache-y! Osobny cache L1 dla instrukcji, osobny dla danych (zwane *icache* oraz *dcache*), osobny dla tablic stron (zwany *TLB* – translation lookaside buffer). Albo lepiej – osobne *TLB* dla danych, osobne dla instrukcji (w końcu używamy go w innych miejscach potoku).

Zyski: możemy skonstruować każdy cache inaczej, stosownie do rodzaju danych, które trzyma.
TLB przechowuje zazwyczaj kilkadziesiąt-kilkaset wpisów; może być mały i *full-associative*.
Cache danych może mieć dwa porty i obsługiwać dwa zapytania jednocześnie (np. jeżeli mamy dwa potoki wykonania).
Cache instrukcji może być z kosmosu.



W myśl idei, iż edukacja na uczelni wyższej powinna być wsteczna, zacofana i nie nadążać za współczesną myślą technologiczną, z dumą prezentujemy

pamięci cache w procesorze AMD Athlon 64 (K8)

Obserwacja: Dekodowanie instrukcji x86 jest bólem w miejscu, gdzie plecy tracą swoją szlachetną nazwę.

Obserwacja: Dekodowanie instrukcji x86 jest bólem w miejscu, gdzie plecy tracą swoją szlachetną nazwę.

Pomysł: Zamiast pamiętać „zakodowane" instrukcje (tak, jak wyemitował je kompilator), pamiętaj zdekodowanie instrukcje (w postaci μ-opsów).

Obserwacja: Dekodowanie instrukcji x86 jest bólem w miejscu, gdzie plecy tracą swoją szlachetną nazwę.

Pomysł: Zamiast pamiętać „zakodowane" instrukcje (tak, jak wyemitował je kompilator), pamiętaj zdekodowanie instrukcje (w postaci μ-opsów). Nazwa: μ-*op cache*.

Obserwacja: Dekodowanie instrukcji x86 jest bólem w miejscu, gdzie plecy tracą swoją szlachetną nazwę.

Pomysł: Zamiast pamiętać „zakodowane” instrukcje (tak, jak wyemitował je kompilator), pamiętaj zdekodowanie instrukcje (w postaci μ-opsów). Nazwa: μ-*op cache*.

Obserwacja: Przewidywanie skoków jest trudne, a daje wielkie korzyści.

Obserwacja: Dekodowanie instrukcji x86 jest bólem w miejscu, gdzie plecy tracą swoją szlachetną nazwę.

Pomysł: Zamiast pamiętać „zakodowane" instrukcje (tak, jak wyemitował je kompilator), pamiętaj zdekodowanie instrukcje (w postaci μ-opsów). Nazwa: *μ-op cache*.

Obserwacja: Przewidywanie skoków jest trudne, a daje wielkie korzyści.

Pomysł: Zamiast cache-ować instrukcje, cache-uj ślad wykonania programu, czyli ciąg instrukcji, które się wykonały, z pominięciem (albo jak kto woli: z uwzględnieniem) skoków.

Obserwacja: Dekodowanie instrukcji x86 jest bólem w miejscu, gdzie plecy tracą swoją szlachetną nazwę.

Pomysł: Zamiast pamiętać „zakodowane" instrukcje (tak, jak wyemitował je kompilator), pamiętaj zdekodowanie instrukcje (w postaci μ-opsów). Nazwa: *μ-op cache*.

Obserwacja: Przewidywanie skoków jest trudne, a daje wielkie korzyści.

Pomysł: Zamiast cache-ować instrukcje, cache-uj ślad wykonania programu, czyli ciąg instrukcji, które się wykonały, z pominięciem (albo jak kto woli: z uwzględnieniem) skoków. Nazwa: *trace cache*.

Dalsze pomysły: Dostęp do cache'a nie musi mieścić się w jednym cyklu; nie chcemy jednak czekać wiele cykli na każdy dostęp. Niech zatem sam cache będzie z-pipeline-owany.

Dalsze pomysły: Dostęp do cache'a nie musi mieścić się w jednym cyklu; nie chcemy jednak czekać wiele cykli na każdy dostęp. Niech zatem sam cache będzie z-pipeline-owany.

Dostęp do cache'a wymaga znajomości adresu fizycznego w pamieci głównej; niestety jest on znany dopiero po wykonaniu mapowania. Ale to mapowanie jest dość prostą operacją: możemy użyć kawałków adresu wirtualnego do odszukiwania danych w cache!

Dalsze pomysły: Dostęp do cache'a nie musi mieścić się w jednym cyklu; nie chcemy jednak czekać wiele cykli na każdy dostęp. Niech zatem sam cache będzie z-pipeline-owany.

Dostęp do cache'a wymaga znajomości adresu fizycznego w pamieci głównej; niestety jest on znany dopiero po wykonaniu mapowania. Ale to mapowanie jest dość prostą operacją: możemy użyć kawałków adresu wirtualnego do odszukiwania danych w cache!

Jeżeli procesor umie wykonywać instrukcje *out-of-order*, to cache miss może wstrzymać wykonanie tylko jednej instrukcji, a nie całego programu; może też odpowiadać na zapytania „dalszych" instrukcji (nazwa: *nonblocking cache*).

Dalsze pomysły: Dostęp do cache'a nie musi mieścić się w jednym cyklu; nie chcemy jednak czekać wiele cykli na każdy dostęp. Niech zatem sam cache będzie z-pipeline-owany.

Dostęp do cache'a wymaga znajomości adresu fizycznego w pamieci głównej; niestety jest on znany dopiero po wykonaniu mapowania. Ale to mapowanie jest dość prostą operacją: możemy użyć kawałków adresu wirtualnego do odszukiwania danych w cache!

Jeżeli procesor umie wykonywać instrukcje *out-of-order*, to cache miss może wstrzymać wykonanie tylko jednej instrukcji, a nie całego programu; może też odpowiadać na zapytania „dalszych" instrukcji (nazwa: *nonblocking cache*).

Predykcja dostępów do cache'a (podobnie jak predykcja skoków w procesorze): możemy „spodziewać się", o który element będzie kolejne zapytanie i optymalizować pod tym kątem (*way prediction*).

Ciekawe problemy, o których nic nie powiem: Co się stanie, jeżeli program zmodyfikuje swój własny kod (dotyczy wszystkich kompilatorów just-in-time, czyli Javy, Javascriptu, .NET, ...). Czy dane w *dcache* rozjadą się z *icache*?

Ciekawe problemy, o których nic nie powiem: Co się stanie, jeżeli program zmodyfikuje swój własny kod (dotyczy wszystkich kompilatorów just-in-time, czyli Javy, Javascriptu, .NET, ...). Czy dane w *dcache* rozjadą się z *icache*? Jak zorganizować dostępy do cache w procesorze wielordzeniowym? Wspólny cache L1? A może rozdzielne L1, ale wspólne L2?

Ciekawe problemy, o których nic nie powiem: Co się stanie, jeżeli program zmodyfikuje swój własny kod (dotyczy wszystkich kompilatorów just-in-time, czyli Javy, Javascriptu, .NET, ...). Czy dane w *dcache* rozjadą się z *icache*? Jak zorganizować dostępy do cache w procesorze wielordzeniowym? Wspólny cache L1? A może rozdzielne L1, ale wspólne L2? Jak zrobić to w komputerze wieloprocesorowym? Co się stanie, jeżeli jeden procesor zapisze dane tylko do swojego cache'a, a drugi procesor będzie chciał je odczytać z pamięci głównej?

Ciekawe problemy, o których nic nie powiem: Co się stanie, jeżeli program zmodyfikuje swój własny kod (dotyczy wszystkich kompilatorów just-in-time, czyli Javy, Javascriptu, .NET, ...). Czy dane w *dcache* rozjadą się z *icache*? Jak zorganizować dostępy do cache w procesorze wielordzeniowym? Wspólny cache L1? A może rozdzielne L1, ale wspólne L2? Jak zrobić to w komputerze wieloprocesorowym? Co się stanie, jeżeli jeden procesor zapisze dane tylko do swojego cache'a, a drugi procesor będzie chciał je odczytać z pamięci głównej? Lepiej: co się stanie, jeżeli oba procesory zapiszą do swoich cache'y?

Ciekawe problemy, o których nic nie powiem: Co się stanie, jeżeli program zmodyfikuje swój własny kod (dotyczy wszystkich kompilatorów just-in-time, czyli Javy, Javascriptu, .NET, ...). Czy dane w *dcache* rozjadą się z *icache*? Jak zorganizować dostępy do cache w procesorze wielordzeniowym? Wspólny cache L1? A może rozdzielne L1, ale wspólne L2? Jak zrobić to w komputerze wieloprocesorowym? Co się stanie, jeżeli jeden procesor zapisze dane tylko do swojego cache'a, a drugi procesor będzie chciał je odczytać z pamięci głównej? Lepiej: co się stanie, jeżeli oba procesory zapiszą do swoich cache'y? Ile kosztuje nas synchronizacja cache'y? Jak jej unikać?

Ciekawe problemy, o których nic nie powiem: Co się stanie, jeżeli program zmodyfikuje swój własny kod (dotyczy wszystkich kompilatorów just-in-time, czyli Javy, Javascriptu, .NET, ...). Czy dane w *dcache* rozjadą się z *icache*? Jak zorganizować dostępy do cache w procesorze wielordzeniowym? Wspólny cache L1? A może rozdzielne L1, ale wspólne L2? Jak zrobić to w komputerze wieloprocesorowym? Co się stanie, jeżeli jeden procesor zapisze dane tylko do swojego cache'a, a drugi procesor będzie chciał je odczytać z pamięci głównej? Lepiej: co się stanie, jeżeli oba procesory zapiszą do swoich cache'y? Ile kosztuje nas synchronizacja cache'y? Jak jej unikać? NUMA: non-uniform memory architecture, czyli nie do każdej pamięci mamy tak samo blisko i łatwo.

Ciekawe problemy, o których nic nie powiem: Co się stanie, jeżeli program zmodyfikuje swój własny kod (dotyczy wszystkich kompilatorów just-in-time, czyli Javy, Javascriptu, .NET, ...). Czy dane w *dcache* rozjadą się z *icache*? Jak zorganizować dostępy do cache w procesorze wielordzeniowym? Wspólny cache L1? A może rozdzielne L1, ale wspólne L2? Jak zrobić to w komputerze wieloprocesorowym? Co się stanie, jeżeli jeden procesor zapisze dane tylko do swojego cache'a, a drugi procesor będzie chciał je odczytać z pamięci głównej? Lepiej: co się stanie, jeżeli oba procesory zapiszą do swoich cache'y? Ile kosztuje nas synchronizacja cache'y? Jak jej unikać? NUMA: non-uniform memory architecture, czyli nie do każdej pamięci mamy tak samo blisko i łatwo. Znany przykład: GPU.

Jak żyć?, czyli co może zrobić programista, aby na świecie zapanował pokój, a jego program był cache-friendly.

Jak żyć?, czyli co może zrobić programista, aby na świecie zapanował pokój, a jego program był cache-friendly.

spatial-locality oraz *temporal-locality* – przykład z pętlami

Jak żyć?, czyli co może zrobić programista, aby na świecie zapanował pokój, a jego program był cache-friendly.

spatial-locality oraz *temporal-locality* – przykład z pętlami
tiling – tekstury na GPU, ale nie tylko

Jak żyć?, czyli co może zrobić programista, aby na świecie zapanował pokój, a jego program był cache-friendly.

spatial-locality oraz *temporal-locality* – przykład z pętlami
tiling – tekstury na GPU, ale nie tylko
data-driven design, czyli architektura gier na konsole

Jak żyć?, czyli co może zrobić programista, aby na świecie zapanował pokój, a jego program był cache-friendly.

spatial-locality oraz *temporal-locality* – przykład z pętlami
tiling – tekstury na GPU, ale nie tylko
data-driven design, czyli architektura gier na konsole
instrukcje *SIMD*

Jak żyć?, czyli co może zrobić programista, aby na świecie zapanował pokój, a jego program był cache-friendly.

- *spatial-locality* oraz *temporal-locality* – przykład z pętlami
- *tiling* – tekstury na GPU, ale nie tylko
- *data-driven design*, czyli architektura gier na konsole
- instrukcje *SIMD*
- *explicit prefetching*

(standing ovation)