**SVEUČILIŠTE U SPLITU**

**FAKULTET ELEKTROTEHNIKE, STROJARSTVA I BRODOGRADNJE**

NAPREDNE ARHITEKTURE RAČUNALA (250)

2

**Toni Biuk**

**Tino Melvan**

**Split, svibanj 2018.**

**SADRŽAJ**

[1. Performance i pristup memoriji 1](#_Toc514619198)

[2. Impakti linija brze memorije 2](#_Toc514619199)

[3. 3](#_Toc514619200)

[4. 4](#_Toc514619201)

[5. 5](#_Toc514619202)

[6. Konzistencija brze memorije 6](#_Toc514619203)

[7. Problem hardvera 8](#_Toc514619204)

[8. Prilog 9](#_Toc514619205)

[8.1. 9](#_Toc514619206)

[8.2. 9](#_Toc514619207)

[8.3. 9](#_Toc514619208)

[8.4. 9](#_Toc514619209)

[8.5. 9](#_Toc514619210)

[8.6. Primjer\_6/src/main.cpp 9](#_Toc514619211)

[8.7. Primjer\_7/src/main.cpp 10](#_Toc514619212)

# Performance i pristup memoriji



# Impakti linija brze memorije

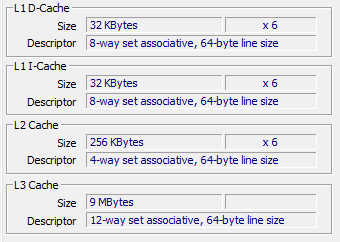
# 

# 

# 

# Konzistencija brze memorije

S problemom konzistencije brza memorija susreće se gotovo na svim više-jezgrenim računalima. To je uzrokovano činjenicom da svaka od jezgara ima svoju zasebnu brzu memoriju (L1 i L2 dio), što se može vidjeti na slici ispod. Na njoj je prikazan ispis programa CPU-Z te prikazuje detaljne podatke o brzoj memoriji za procesor koji se nalazi unutar računala .



Slika 6.1. Ispis programa CPU-Z

Problem možemo demonstrirati programom danim u prilogu (8.6). Ako ga pokrenemo dobijemo ispis dan u nastavku (Slika 6.2).



Slika 6.2. Ispis primjera 6

Razlog ovakvog ponašanja direktno je povezan s duljinama linije brze memorije i ponašanjem brzih memorija kada se radi o višenitnosti. Prvo, bitno je napomenuti da su elementi niza alocirani dinamički spremljeni redom u memoriji, jedan iza drugoga. S obzirom na to da brza memorija u korištenoj mašini koristi 64-bajtne linije, u svaku od njih može se spremiti najviše šesnaest 32-bitnih cijelih brojeva (engl. *Integer*). To znači da, u prvom našem slučaju, gdje pristupamo elementima s pozicijama 0, 1, 2 i 3 u nizu, imamo veliku vjerojatnost da će svi traženi elementi biti na istoj liniji brze memorije. S druge strane, u drugom slučaju, gdje pristupamo elementima s pozicijama 16, 32, 48 i 64, imamo obrnutu situaciju kada imamo veliku vjerojatnost da su svi elementi na različitoj liniji brze memorije. Razlog zašto je to toliko bitno jest, kada jezgra mijenja vrijednost u svojoj brzoj memoriji, također onemogućuje svim drugim jezgrama korištenje stare vrijednosti za odgovarajuću adresu. Nadalje, kada se onemogućuje nešto u broj memoriji, onemogućuje se cijela linija, a ne samo njezin dio. To znači da, niti jedna jezgra, u svom sljedećem pristupu, neće pronaći traženu vrijednost za adresu u brzoj memoriji, što rezultira u mnogo sporijem izvršavanju.

# Problem hardvera

Ovaj primjer trebao bi demonstrirati čudno ponašanje brze memorije kao posljedica ovisnosti o drugom hardveru. Iako u potpunosti znamo kako bi se brza memorija trebala ponašati, činjenica je da je i dalje teško predvidjeti. Pogledom na primjer dan u prilogu (8.7), moglo bi se krivo zaključiti da će se funkcija koja inkrementira manje varijabli (WeirdnessAC) uvijek (ili barem u prosijeku) izvršavati brže od funkcije koja inkrementira više varijabli (WeirdnessABCD, WeirdnessACEG), naravno, ako su one po svemu ostalome identične. Puštanje sveukupnog programa otkriti će, pak, da to nije slučaj. Inkrement funkcija WeirdnessABCD i WeirdnessACEG uvijek su blizu, što se tiče vremena izvršavanja. Treća funkcija, WeirdnessAC, s druge strane, pomalo je nepredvidiva. Kolika puta se pojavila kao najbrža (Slika 7.1), toliko puta se pojavila i kao najsporija (Slika 7.2).



Slika 7.1. Brzi AC inkrement



Slika 7.2. Spori AC inkrement

Razlozi zašto bi ovo mogao biti slučaj su razni. Najlakše bi bilo zaključiti kako ovo ima veze s redoslijedom kojim instrukcije završe na procesoru, ili pak s činjenicom koliko se brzo otkrije podudaranje u brzoj memoriji. Problem može biti i dublji, ako kompajler (tj. procesor) koristi SSE (engl. *Streaming SIMD Extensions*) instrukcije. Takva vrsta ne radi s jednom po jednom varijablom, već ih kupi i do četiri odjednom, ukoliko su instrukcije dovoljno slične.

# Prilog

## Primjer\_1/src/main.cpp

#include <Utility/Clock.hpp>

#include <Utility/Traces.hpp>

#include <cstdint>

#include <cstdlib>

#define BIT (1U)

#define BYTE (8U \* BIT)

#define KILOBYTE (1024U \* BYTE)

#define MEGABYTE (1024U \* KILOBYTE)

#define ARR\_MEMORY\_SIZE (256U \* MEGABYTE)

#define ARR\_LENGTH (ARR\_MEMORY\_SIZE / (sizeof(int32\_t) \* BYTE))

int32\_t main() {

Clock clock;

auto \*arr = new int32\_t[ARR\_LENGTH]();

clock.Start();

for (auto i = 0U; i < ARR\_LENGTH; i++) {

arr[i] \*= 3;

}

INFO("Loop\_1: %lld ms\n", clock.ElapsedMiliSeconds());

clock.Start();

for (auto i = 0U; i < ARR\_LENGTH; i += 16) {

arr[i] \*= 3;

}

INFO("Loop\_2: %lld ms\n", clock.ElapsedMiliSeconds());

delete[] arr;

return EXIT\_SUCCESS;

}

## Primjer\_2/src/main.cpp

#include <Utility/Clock.hpp>

#include <Utility/Traces.hpp>

#include <cstdint>

#include <cstdlib>

#define BIT (1U)

#define BYTE (8U \* BIT)

#define KILOBYTE (1024U \* BYTE)

#define MEGABYTE (1024U \* KILOBYTE)

#define ARR\_MEMORY\_SIZE (256U \* MEGABYTE)

#define ARR\_LENGTH (ARR\_MEMORY\_SIZE / (sizeof(int32\_t) \* BYTE))

#define INCREMENT (1024U)

int32\_t main() {

Clock clock;

auto \*arr = new int32\_t[ARR\_LENGTH]();

const auto K = INCREMENT;

clock.Start();

for (auto i = 0U; i < ARR\_LENGTH; i += K) {

arr[i] \*= 3;

}

INFO("Loop: %lld ms\n", clock.ElapsedMiliSeconds());

delete[] arr;

return EXIT\_SUCCESS;

}

## 

## 

## 

## Primjer\_6/src/main.cpp

#include <Utility/Clock.hpp>

#include <Utility/Traces.hpp>

#include <cstdint>

#include <cstdlib>

#include <thread>

#include <vector>

#define ARR\_LENGTH (1024U)

#define NUM\_OF\_THREADS (4U)

auto \*s\_counter = new int32\_t[ARR\_LENGTH]();

static void UpdateCounter(const uint32\_t position) noexcept {

for (auto j = 0U; j < 100000000U; j++) {

s\_counter[position] = s\_counter[position] + 3;

}

}

int32\_t main() {

Clock clock;

const uint32\_t positions1[] = { 0, 1, 2, 3 };

const uint32\_t positions2[] = { 16, 32, 48, 64 };

std::vector<std::thread> threads;

threads.reserve(NUM\_OF\_THREADS);

clock.Start();

for (auto i = 0U; i < NUM\_OF\_THREADS; i++) {

threads.emplace\_back(std::thread(UpdateCounter, positions1[i]));

}

for (auto& thread : threads) {

thread.join();

}

INFO("Positions\_1 took %.2f seconds\n", clock.ElapsedMiliSeconds() / 1000.0);

threads.clear();

clock.Start();

for (auto i = 0U; i < NUM\_OF\_THREADS; i++) {

threads.emplace\_back(std::thread(UpdateCounter, positions2[i]));

}

for (auto& thread : threads) {

thread.join();

}

INFO("Positions\_2 took %.2f seconds\n", clock.ElapsedMiliSeconds() / 1000.0);

return EXIT\_SUCCESS;

}

## Primjer\_7/src/main.cpp

#include <Utility/Clock.hpp>

#include <Utility/Traces.hpp>

#include <cstdint>

#include <cstdlib>

uint32\_t A = 0;

uint32\_t B = 0;

uint32\_t C = 0;

uint32\_t D = 0;

uint32\_t E = 0;

uint32\_t F = 0;

uint32\_t G = 0;

static void WeirdnessABCD() noexcept {

for (auto i = 0U; i < 200000000U; i++) {

A++;

B++;

C++;

D++;

}

}

static void WeirdnessACEG() noexcept {

for (auto i = 0U; i < 200000000U; i++) {

A++;

C++;

E++;

G++;

}

}

static void WeirdnessAC() noexcept {

for (auto i = 0U; i < 200000000U; i++) {

A++;

C++;

}

}

int32\_t main() noexcept {

Clock clock;

clock.Start();

WeirdnessABCD();

auto elapsedTimeMS = clock.ElapsedMiliSeconds();

INFO("A++; B++; C++; D++;\t\t%lld ms\n", elapsedTimeMS);

clock.Start();

WeirdnessACEG();

elapsedTimeMS = clock.ElapsedMiliSeconds();

INFO("A++; C++; E++; G++;\t\t%lld ms\n", elapsedTimeMS);

clock.Start();

WeirdnessAC();

elapsedTimeMS = clock.ElapsedMiliSeconds();

INFO("A++; C++;\t\t\t%lld ms\n", elapsedTimeMS);

return EXIT\_SUCCESS;

}