**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](#_Toc514702284)

[1.1. Kod 1](#_Toc514702285)

[1.2. Rezultati 1](#_Toc514702286)

[2. Impakti linija brze memorije 3](#_Toc514702287)

[2.1. Kod 3](#_Toc514702288)

[2.2. Rezultati 3](#_Toc514702289)

[3. L1 i L2 brza memorija 5](#_Toc514702290)

[3.1. Kod 5](#_Toc514702291)

[3.2. Rezultati 5](#_Toc514702292)

[4. 7](#_Toc514702293)

[5. 8](#_Toc514702294)

[6. Konzistencija brze memorije 9](#_Toc514702295)

[6.1. Kod 9](#_Toc514702296)

[6.2. Rezultati 10](#_Toc514702297)

[7. Problem hardvera 12](#_Toc514702298)

[7.1. Kod 12](#_Toc514702299)

[7.2. Rezultati 13](#_Toc514702300)

[8. Prilog 14](#_Toc514702301)

[8.1. Primjer\_6/src/main.cpp 14](#_Toc514702302)

# Performance i pristup memoriji

## Kod

#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;

}

## Rezultati

Dani kod izvršava dvije petlje te mjeri vrijeme potrebno da se one izvrše. Prva prolazi kroz svaki element danog niza, dok druga kroz svaki šesnaesti. Druga petlja, dakle, izvršava šesnaest puta manje prolazaka kroz petlju, pa bi za očekivati bilo da će ona i mnogo kraće trajati. Međutim, kada vrijeme zapravo izmjerimo (Slika 1.1) vidimo da to nije slučaj. U danom primjeru, druga petlja je svega 10% brža. Razlog je sporost pristupanja memoriji, kako bi se dohvatio element niza, dok je sama aritmetička operacija zanemariva (vremenski govoreći).



Slika 1.1. Ispis primjera 1

S obzirom na to da program ne mora direktno pristupati glavnoj memoriji, već umjesto toga pristupa brzoj, gdje je prvim pristupom učitan cijeli niz, svo vrijeme utrošeno na to postaje mnogo kraće.

# Impakti linija brze memorije

## Kod

#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;

}

## Rezultati

Slika 2.1. Ovisnost veličine koraka o vremenu izvršavanja koda u "primjeru 2"

U ovom primjeru ispituje se brzina izvršavanja petlje ovisno o veličini koraka. Za broj koraka manjih od 16, vrijeme izvršavanja je relativno slično, iako se uočava mali pad zbog manjeg broja operacija koje je potrebno izvršiti. Vrijeme pristupa brzoj memoriji za takve korake je konstantno jer brza memorija posjeduje linije od 64 bajta. Također, upravo zato što ima linije od 64 bajta, kod koraka većih od 16 uočit će se pad. To se događa zbog činjenice da brza memorija može spremiti točno 16 elemenata u svoju jednu liniju (*int32\_t* velik je 4 bajta, pa vrijedi 16 \* 4bajta = 64bajta). S daljnjim porastom koraka, program više neće morati dirati sve linije brze memorije, već će određene moći preskakati, što će uzrokovati velika ubrzanja.

# L1 i L2 brza memorija

## Kod

#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 (8U \* MEGABYTE)

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

int32\_t main() {

Clock clock;

const auto steps = 64U \* 1024U \* 1024U;

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

INFO("Array size: %llu\n", ARR\_LENGTH);

clock.Start();

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

arr[(i \* 16U) % ARR\_LENGTH]++;

}

const auto elapsedNS = clock.ElapsedNanoSeconds();

INFO("Time per element: %lld ns\n", elapsedNS / steps);

delete[] arr;

return EXIT\_SUCCESS;

}

## Rezultati

Mijenjajući veličinu ulaznog niza cijelih brojeva (ARR\_MEMORY\_SIZE), iz prethodno danog programa dobije se graf prikazan u nastavku.

Slika 3.1. Ovisnost veličine niza i vremena izvršavanja po elementu niza u primjeru 3.

Promatranjem ovisnosti niza i vremena izvršavanja možemo odrediti veličinu brze memorije i njenih dijelova. Iz grafa, očite promijene nastupaju poslije 32kB i 9MB. Iz toga se može iščitati da je L1 velik 32kB, dok je L3 velik 9MB. Ako pak ne znamo da procesor ima L3 brzu memoriju, dalo bi se zaključiti da je L2 velik 9MB, te da L3 ne postoji. Međutim, to nije slučaj, već samo L1 i L2 imaju slične performanse u ovome slučaju, a i vrijednosti su zaokruživane na donji cijeli broj. L2 bi trebao biti velik 256kB. Ako podatke nije moguće smjestiti u L3 brzu memoriju, vrijeme izvršavanja počet će brzo rasti zbog brojnih pristupanja glavnoj memoriji, što je vidljivo iz grafa.

# Razine paralelizma

## Kod

#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\_LENGTH (2U)

int32\_t main() {

Clock clock;

const auto steps = 32U \* MEGABYTE;

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

clock.Start();

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

arr[0U]++;

arr[0U]++;

}

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

clock.Start();

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

arr[0U]++;

arr[1U]++;

}

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

delete[] arr;

return EXIT\_SUCCESS;

}

## Rezultati

Rezultati izvršavanja prethodno danog koda, uz isključenu optimizaciju kompajlera, dani su na slici ispod.



Slika 4.1. Rezultati izvršavanja primjera 4.

Druga petlja brža je od prve zbog činjenice da moderni procesori mogu pristupati višestrukim lokacijama unutar L1 brze memorije istovremeno. Prva petlja ovakvo nešto ne može iskorištavati zbog toga što pristupa istom elementu, pa paralelizacija nije moguća zbog ovisnosti između više operacija. S druge strane, ovakav tip paralelizacije urednu prolazi kod petlje broj dva, zbog toga što se radi o dva različita elementa.

# 

# Konzistencija brze memorije

## Kod

#include <Utility/Clock.hpp>

#include <Utility/Traces.hpp>

#include <cstdint>

#include <cstdlib>

#include <thread>

#include <vector>

#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 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();

}

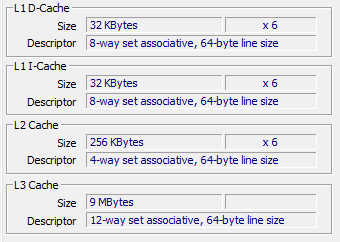
INFO("Positions\_2 took %.2f seconds\n", clock.ElapsedMiliSeconds() / 1000.0);

return EXIT\_SUCCESS;

}

## Rezultati

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 danim programom. Ako ga pokrenemo (bez ikakvih optimizacija) 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

## Kod

#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;

}

## Rezultati

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 poglavlju 7.1, 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. Iz priloženog rezultata (SLIKA) može se vidjeti da je WeirdnessABCD duplo sporija od WeirdnessACEG, a treća funkcija, WeirdnessAC, ima gotovo jednako vrijeme izvršavanja kao i prethodno spomenuta.



Slika 7.1. Ispis primjera 7

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. Sve u svemu, WeirdnessABCD je najsporija vjerojatno zbog činjenice što se tada podaci još ne nalaze u broj memoriji. Što se tiče razlike između WeirdnessACEG i WeirdnessAC, ona je slična zbog situacije kao i u prvom primjeru (1. Performance i pristup memoriji).

Zanimljiva je i činjenica da moderni kompajleri ovakve situacije jako dobro mogu optimizirati, pa tako GCC kompajler uz optimizaciju -O3 ima gotovo zanemarivo izvršavanje ovog koda.



Slika 7.2. Primjer 7 uz 03 optimizaciju GCC kompajlera

# Prilog