Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Bakalářská práce

Objektivní analýza výkonu překladačů vybraných programovacích jazyků

Plzeň 2020 Jan Pizúr

Místo této strany bude zadání práce.

Prohlášení

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 21. července 2020

Jan Pizúr

Abstract

The aim of this thesis is to introduce modern techniques of objective evaluation of compilers performance. Another aim is to design set of non-trivial testing tasks. Those tasks will be realized through programming languages C, C++, Java and Object Pascal. Afterwards they will be compiled by some of available compilers and they will be compared by final codes of chosen compilers outputs.

The first part of the thesis is dealing with profiling problems and methods of data collection which are used by modern tools intended for application performance measuring.

The second part of the thesis includes design of testing methods, description of used measuring, statistics of achieved outputs and overall evaluation of outputs.

Abstrakt

Hlavním cílem této práce je seznámit se s moderními technikami objektivního posuzování výkonu překladačů programovacích jazyků, dále navrhnout sadu netriviálních testovacích úloh, tyto úlohy realizovat v programovacích jazycích C, C++, Java a Object Pascal, přeložit je několika dostupnými překladači a porovnat pomocí výsledných kódů výkony zvolených překladačů.

První část této práce se důkladně zabývá problematikou profilování a metodami sběru dat, které jsou používány moderními nástroji určenými k měření výkonu aplikací.

Druhá část práce obsahuje návrh testovacích úloh, popis použitého způsobu měření, statistiky dosažených výkonů a také celkové zhodnocení výsledků.

Poděkování

Touto cestou bych rád poděkoval Ing. Kamilu Ekšteinovi, Ph.D. za ochotu a cenné rady při vedení bakalářské práce.

Obsah

1 Úvod

V dnešní době je kladen obrovský důraz na rychlost vývoje nejrůznějších aplikací. Z toho důvodu jsou stále častěji používány programovací jazyky, které poskytují různé metody usnadňující vývoj aplikací, tedy například automatickou správu paměti. Použití těchto metod dělá daný programovací jazyk jednodušším, člověku lépe srozumitelným, a tedy urychlujícím samotný vývoj aplikace, jelikož mohou být zdrojové kódy kratší, čitelnější a je zde menší pravděpodobnost vzniku chyb.

Příkazy programovacích jazyků zapsaných ve zdrojovém kódu nicméně procesor zpracovat nedokáže, jelikož zná pouze vlastní sadu strojových instrukcí, které jsou zapsány ve formě čísel, tedy pro člověka velice špatně čitelné. Existují však speciální programy, souhrnně označovány jako překladače, které slouží pro překlad zdrojového kódu do mezikódu nebo kódu strojového. Zatímco strojový kód je zpracován přímo procesorem, mezikód je zpracován virtuálním strojem, který zajišťuje převod na strojové instrukce.

Překladače provádí kromě samotného překladu také řadu optimalizací, které mohou významně ovlivnit dobu běhu, paměťovou náročnost, rychlost překladu nebo velikost vyprodukovaného kódu. Výkony jednotlivých překladačů se však značně liší, cílem této práce je proto seznámit se s moderními technikami posuzování výkonu překladačů, navrhnout sadu vhodných algoritmů pro objektivní analýzu výkonu překladačů programovacích jazyků *C*, *C++*, *Java* a *Object Pascal*, následně tyto algoritmy co nejpodobněji implementovat ve zmíněných jazycích, přeložit několika různými překladači a detailně porovnat dosažené výkony.

2 Výkon překladače

Před samotným zkoumáním výkonu překladačů je nejprve nezbytně nutné definovat, jak bude výkon překladače v celém textu chápán, jelikož se nejedná o ustálený pojem, a proto může být dvěma různými lidmi chápán výkon překladače zcela odlišně. Každý programátor může mít totiž jiné nároky na překladač. Zatímco jeden programátor může požadovat co nejrychlejší vykonání vyprodukovaného kódu, jiný programátor může požadovat nízké paměťové nároky i za cenu značně pomalejšího kódu, a to například z důvodu omezené velikosti dostupné paměti. V této práci mají být zkoumány tři parametry – doba běhu, paměťová náročnost a čas nutný k překladu zdrojového kódu. Jelikož nemá žádný smysl kombinovat tyto parametry do jedné agregované hodnoty, která by reprezentovala celkový výkon překladače, budou tyto parametry zkoumány odděleně a nejvýkonnějším překladačem bude prohlášen ten, který bude v dané oblasti vykazovat nejlepší výsledky. Teoreticky se tedy může stát, že bude překladač v jedné oblasti nejvýkonnějším a v druhé naopak nejméně výkonným. Takto odděleně zkoumají jednotlivé parametry také vývojáři známého překladače *Visual C++*, viz [18].

Měření výkonnostních parametrů se obecně nazývá *profilování*, které zahrnuje celou škálu technik, jakými mohou být tyto parametry měřeny. Kromě toho jsou dnešní procesory vybaveny hardwarovými jednotkami, které slouží k měření nejrůznějších výkonnostních parametrů a jsou velice často využívány při *profilování*.

3 Hardwarové výkonnostní čítače

Hardwarové výkonnostní čítače jsou registry poskytované moderními procesory, které slouží k počítání nízkoúrovňových výkonnostních parametrů, souhrnně označovaných jako *události* [11, 14, 16]. Tyto čítače se nachází v jednotce nazývané *Performance Monitoring Unit (PMU)* [6, 11]. Jelikož jsou měření prováděna na hardwarové úrovní, jsou režijní náklady poměrně nízké. Data poskytovaná těmito čítači jsou nejčastěji používána právě k měření výkonu. Implementace těchto registrů ovšem nejsou jednotné a mohou se lišit jednak v počtu dostupných čítačů a jednak v seznamu událostí, které mohou být na daném procesoru měřeny [11, 14]. Jelikož procesor vlastní jen velice omezený počet zmíněných čítačů, ve srovnání s počtem měřitelných událostí, může být poměrně složité vybrat, které události měřit. Běžně dostupnými událostmi jsou například:

- počet hodinových cyklů,
- počet provedených instrukcí,
- statistika mezipamětí a hlavní paměti,
- statistika predikce větvení a podobně [14].

V případech, kdy je nezbytné počítat větší množství událostí než je počet dostupných čítačů, je nutné program spustit vícekrát nebo využít multiplexování. Při použití multiplexování není událost měřena po celou dobu měření, ale jen určitou část, aby bylo možné změřit vícero událostí. Čítače jsou v takovém případě periodicky přeprogramovávány, čímž je zajištěno počítání několika událostí jedním čítačem. Na konci měření je celkový počet událostí vynásoben podílem celkové doby měření a doby, během které byla událost měřena [7]. Tento způsob však snižuje přesnost naměřených dat, která závisí na počtu událostí, které jsou jednim čítačem měřeny. Tedy čím více událostí bude měřeno jedním čítačem, tím nepřesnější výsledky budou [7, 14]. Hardwarové výkonnostní čítače mohou pracovat ve dvou režimech:

- *režim počítání* (angl. *counting mode*), ve kterém je počítán celkový počet událostí,
- *vzorkovací režim* (angl. *statistical sampling mode*), který slouží k vzorkovacímu profilování na základě přetečení čítače [11, 14].

Oba tyto režimy mají své uplatnění při měření výkonu a mají své výhody i nevýhody. Kromě toho může být jeden režim emulován druhým, a to v případech, kdy není požadovaný režim dostupný na dané platformě. Například na platformách, které nepodporují hardwarové přerušení při přetečení čítače, může být použito hardwarové přerušení časovače (*IRQ 0*) pro periodickou kontrolu přetečení. Naopak na platformách, které podporují pouze vzorkovací profilování, mohou být hodnoty postupně sečteny, čímž je získána jedna agregovaná hodnota. Nicméně takovéto emulace mohou mít neblahý vliv na přesnost výsledků [14].

3.1 Režim počítání

Režim počítání je často používán k nalezení událostí, které způsobují pomalý běh programu, jelikož je hodnota čítače přečtena na začátku a konci měření, čímž je získán celkový počet výskytů měřených událostí. Pokud tedy program využívá například mezipaměť neefektivním způsobem, kdy dochází k velkému množství výpadků vyrovnávacích pamětí (angl. cache miss), je možné využít tento režim a odhalit daný problém. Není ovšem možné zjistit, které části kódu mohou za výskyt naměřených událostí. Z toho důvodu je tento režim vhodné použít především v prvotních fázích výkonnostní analýzy programu [11, 14].

3.2 Vzorkovací režim

Je-li nutné například zjistit, které části kódu mohou za výskyt naměřených události, je vhodné použít vzorkovací režim, ve kterém je generováno přerušení při přetečení daného čítače, čímž je vyvolána obsluha přerušení, během které mohou být získány informace o zařízení v době obsluhy. Tyto informace mohou obsahovat například hodnotu instrukčního ukazatele, čímž je možné událost přiřadit přímo k instrukci, která ji vyvolala [11, 14].

3.3 Realizace PMU

Realizace PMU se liší nejen napříč výrobci procesorů, ale také napříč různými architeturami procesorů stejného výrobce [14]. Například na procesorech značky *Intel*, s architekturou *Nehalem*, je *PMU* realizována pomocí skupiny tzv. *Model Specific registrů*, které slouží jednak ke konfiguraci jednotky a jednak k počítání jednotlivých událostí. Tato skupina obsahuje tři

fixní [4, 11] a čtyři programovatelné čítací registry, čtyři řídící registry a několik dalších registrů, které slouží například k zakázání některých registrů, k indikaci přetečení registrů apod [4]. Fixní čítací registry slouží k počítání předdefinovaných událostí, kterými jsou počet odbavených instrukcí, počet provedených hodinových cyklů a počet provedených referenčních cyklů [11]. Programovatelné čítací registry slouží taktéž k počítání událostí, nicméně může být u těchto registrů vybrána událost, která má být monitorována. Řídící registry programovatelných čítačů slouží k jejich konfiguraci a je pomocí nich možné určit, která událost má být sledována, v jaké úrovni privilegovaného režimu, zda má být vyvoláno přerušení při přetečení čítače apod [4, 11]. Jak již bylo zmíněno, realizace *PMU* se liší, princip je ale stejný i na jiných architekturách procesorů [14].

4 Profilování

Profilování je forma dynamické analýzy programu, během které jsou naměřena různá data, která mohou poskytnout vývojáři cenné informace o dané aplikaci, například jaká je doba běhu aplikace, jaké jsou paměťové nároky, kolik času zabralo vykonání konkrétní metody (funkce, procedury), zda došlo k vysokému počtu výpadků cache pamětí apod. Na základě těchto dat je například možné určit, ve kterých částech programu bylo stráveno nejvíce času (angl. *hotsposts*), nebo detekovat některé chyby, které programátor do aplikace během vývoje zanesl – například neuvolnil dříve alokovanou paměť.

Tato data jsou získávána různými metodami, které s sebou přináší různé výhody, ale také své nevýhody. Jak již bylo zmíněno v předchozím odstavci, naměřená data mohou poskytnout vývojáři cenné informace, ale pouze mohou, nemusí. Získaná data mohou být totiž značně zkreslena zvolenou měřící metodou, což může vést například k tomu, že bude programátor optimalizovat části kódu, které nejsou důvodem pomalého běhu aplikace. To je samozřejmě nežádoucí a je tedy právě na programátorovi, aby se s metodami seznámil a dokázal data správně interpretovat a případně odhalil nedostatky v konkrétních měřeních.

Většina programátoru se minimálně s jednoduchou metodou profilování již setkala, aniž by si to třeba uvědomila. Téměř každý totiž využil některou z metod určených k měření času, poskytovaných standardními knihovnami programovacích jazyků. Takovýto způsob profilování se nazývá *manuální instrumentace*. Kromě *manuální instrumentace* lze použít také nástroje určené přímo k profilování, takzvané *profilery*, které využívají sofistikovanější metody sběru dat.

4.1 Manuální instrumentace

Manuální instrumentace je konkrétní typ jedné z metod sběru dat, takzvané *instrumentace*, která je detailně popsána v další kapitole. Ve zkratce se ale jedná o metodu, během které jsou do programu vkládány speciální měřící instrukce, které narušují klasický chod aplikace. Manuálním typem *instrumentace* je potom nazýván konkrétní typ, kdy jsou do programu tyto instrukce vkládány manuálně, a to přímo vývojářem.

Ačkoliv lze manuální instrumentaci využít k měření různých výkonnostních parametrů, kterými mohou být například počet volání jednotlivých

funkcí, počet provedených alokací paměti apod., je tato metoda často využívána pro měření doby běhu různých částí kódu.

Prvním způsobem manuální instrumentace, při které je měřena doba běhu, je využití tzv. *high-resolution performance counteru*, který se nachází přímo v procesoru. Konkrétně se jedná o *64-bitový* registr známý jako *Time Stamp Counter (TSC)* [19]. V dnešní době je tento registr využíván za určitých podmínek téměř všemi metodami, které slouží k měření tzv. *high-resolution timestamps*, tedy velice přesných časových vzorků. Kromě přímého čtení hodnoty z registru *TSC* je možné využít funkce poskytované operačními systémy, tedy například *QueryPerformanceCounter()* na operačním systému *Windows*, nebo *clock_gettime()* na operačním systému *GNU/Linux*. Další možností je využít funkce poskytované standarními knihovnami programovacích jazyků, případně využít knihovny třetích stran.

4.1.1 Time Stamp Counter

Time Stamp Counter (TSC) je *64-bitový* registr, který udává počet provedených hodinových cyklů od resetu procesoru [11, 20] a k němuž je možné přistupovat s nízkou latencí a nízkými režijními náklady. U vícejádrových procesorů má každé jádro vlastní *TSC* registr, což s sebou ale přináší řadu potenciálních problémů [3, 15, 19].

První problém může nastat v případě, kdy nejsou frekvence jednotlivých *TSC* registrů napříc jádry procesoru sesynchronizovány [15, 19]. Pokud totiž kód vlákna nebude během měření zpracováván pouze jedním jádrem procesoru, bude moci nastat případ, ve kterém vyjdou chybné výsledky. Ty budou zapříčiněny přečtením hodnot z rozdílných *TSC* registrů. K získání doby běhu je totiž nutné přečíst hodnotu z registru dvakrát, a to na začátku měřeného bloku, kdy je získána první přečtená hodnota x_1 a na konci měřeného bloku, kdy je získána druhá hodnota x_2. Následně je nutné vypočítat rozdíl techto dvou hodnot, tedy $x_2 - x_1$, čímž je získána výsledná doba běhu. Pokud ale nebudou obě instrukce čtecí hodnotu z *TSC* registru zpracovány stejným jádrem procesoru, což samozřejmě reálně může nastat, bude výsledek chybný, jelikož obecně *TSC* registry jednotlivých jader sesynchronizovány nejsou. Tento problém by mohl být vyřešen například nastavením afinity procesu tak, aby byl proces zpracováván pouze jedním jádrem procesoru. V praxi by ale takové řešení mohlo být nevhodné, jelikož by zejména u vícevláknových aplikací došlo k obrovskému poklesu výkonu, a to ve smyslu značného zvýšení doby běhu, což je samozřejmě nežádoucí [15].

Druhý problém spočívá v závislosti frekvence *TSC* na frekvenci daného jádra. Je nutné si uvědomit, že procesory mohou měnit svou frekvenci, a

to například při přechodu do *úsporného režimu* (tzv. *Power Saving Mode*), ve kterém je frekvence snížena, nebo při využití technologie *Turbo Boost*, během které je naopak frekvence procesoru zvýšena. Pokud tedy bude frekvence *TSC* opravdu záviset na frekvenci příslušného jádra a dojde během měření k jakékoliv změně frekvence, nebude možné *TSC* využít pro měření reálného času (angl. *wall-clock time*), jelikož by výsledky získané tímto zdrojem času byly chybné [15]. Hodnota, která je přečtena z *TSC* registru totiž udává počet hodinových cyklů provedených od resetu procesoru, nikoliv počet sekund, milisekund apod. Výsledky jsou tedy vždy v hodinových cyklech a pro převod na jednotku času je nutné znát frekvenci *TSC*, kterou musí být výsledek vydělen. Pokud však nebyla během měření tato frekvence konstantní, bude po převodu výsledek chybný [10, 15].

Výrobci procesorů si ale těchto nedostatků byli vědomi, a proto jsou v novějších procesorech (od mikroarchitektury *Nehalem* z roku 2008) zavedeny invariantní verze *TSC* registrů. Tato verze *TSC* registrů zaručuje inkrementaci se stejnou frekvencí nezávisle na *Turbo Boost* technologii a všech *ACPI P-*, *T-* a *C-stavech* procesoru [20]. V dnešní době se navíc operační systém snaží při bootování *TSC* sesynchronizovat napříč všemi jádry – synchronizace ale není garantována [19].

Hodnotu *TSC* registru lze přečíst pomocí instrukce *RDTSC*, která tuto hodnotu načte do registrů *EDX:EAX*, přičemž je do registru *EAX* uloženo 32 méně významných bitů a do registru *EDX* 32 zbylých bitů. Jelikož jsou hodnoty v těchto registrech přepsány, musí být před voláním instrukce *RDTSC* tyto hodnoty uloženy do zásobníku a následně uloženy zpět do registrů [11, 15].

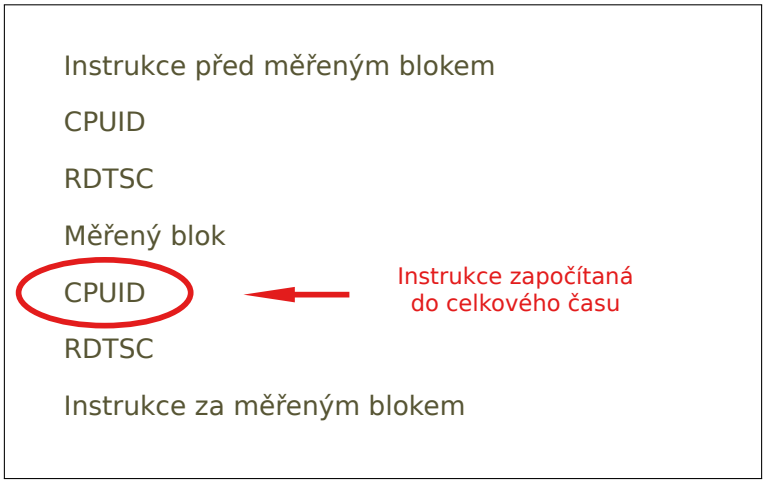
Dnešní procesory podporují metodu zpracování instrukcí mimo své pořadí (angl. *Out of Order execution*), při níž jsou instrukce zpracovávány mimo pořadí, které uvádí program uložený v operační paměti. Tato metoda však způsobuje komplikace během měření. Programátor, který bude měřit určitý úsek kódu může předpokládat, že bude instrukce *RDTSC* provedena přesně na začátku a konci měřeného bloku, tento předpoklad by byl ale chybný. Jelikož jsou instrukce zpracovávány mimo své pořadí, může být totiž během měření provedena instrukce, která se v měřeném bloku nenachází, což ovlivní celkový výsledek. Tomuto problému lze předejít voláním *serializační instrukce*[1] těsne před instrukcí *RDTSC*, čímž bude procesor nucen vykonat veškeré předchozí instrukce a až poté bude moci dále pokračovat ve vykonávání zbylých instrukcí, čímž bude zajištěno změření správné části kódu – tedy že budou během měření vykonány pouze instrukce nacházející se v

[1] Instrukce, která nutí procesor vykonat veškeré předchozí instrukce [15]

měřeném bloku a že nebudou tyto instrukce vykonány mimo měřený blok [11, 15].

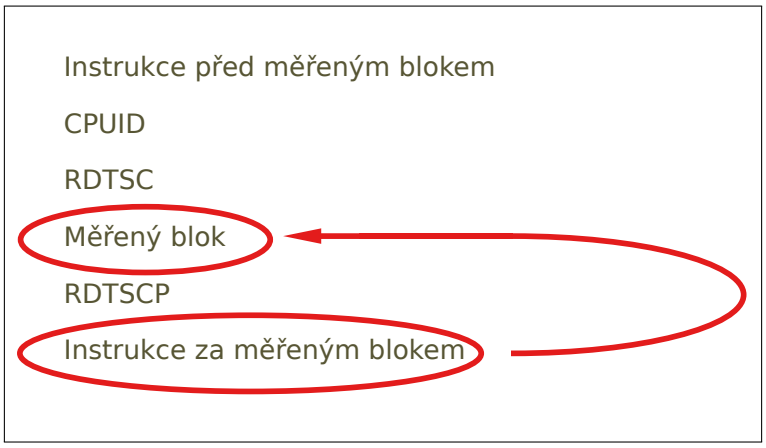
Jednou ze serializačních instrukcí je například instrukce *CPUID*, kterou je jednak možné vykonat v jakémkoliv režimu procesoru a jednak nemá vliv na chod programu, tedy kromě modifikace registrů *EAX*, *EBX*, *ECX* a *EDX*. Při použití instrukce *RDTSC* zároveň s instrukcí *CPUID* už tedy nestačí uložit do zásobníku pouze hodnoty z registrů *EAX* a *EBX*, jako při samostatném použití *RDTSC*, ale také hodnoty z dvojice registrů *ECX*, *EDX* a následně tyto hodnoty uložit zpět do registrů [11, 15].

Nyní by se mohlo zdát, že stačí volat instrukci *CPUID* před každým voláním *RDTSC* a metoda *Out of Order execution (OOE)* neovlivní měřený blok. Metoda sice měřený blok opravdu neovlivní, jak je ale vidět na obrázku 4.1, do celkového počtu provedených cyklů bude započítána i doba nutná k provedení instrukce *CPUID* [15].



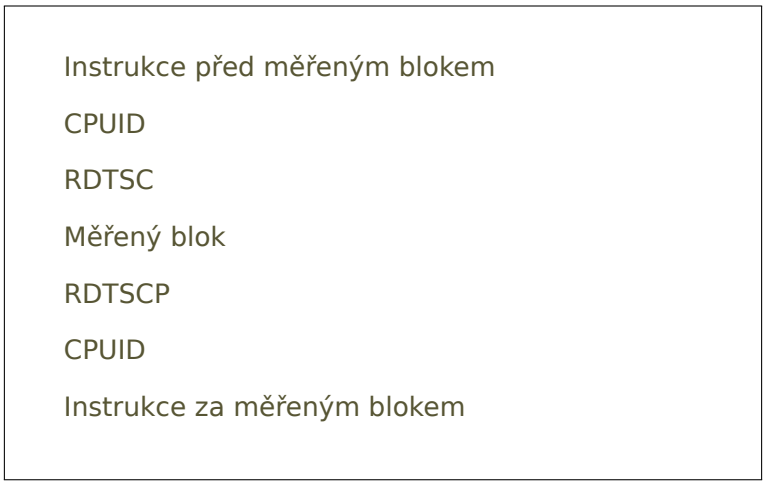
Obrázek 4.1: Příklad použití instrukce *CPUID*, která je započítána do změřeného času.

Z toho důvodu je nutné změřit režijní náklady dané instrukce a následně je odečíst od celkového výsledku, nebo použít novější instrukci *RDTSCP* [15]. Instrukce *RDTSCP*, která byla přidána do novějších procesorů jakožto serializační verze instrukce *RDTSC*, kromě přečtení hodnoty z *TSC* registru přečte také *ID* procesoru, přičemž je do registrů *EDX* a *EAX* uložena hodnota *TSC* registru a do registru *ECX* *ID* procesoru. Nutno zmínit, že instrukce *RDTSCP* není klasickou serializační instrukcí, jelikož poskytuje pouze „pseudo“ serializaci, což znamená, že je procesor nucen vykonat veškeré předešlé instrukce, ale zároveň můžou být procesorem vykonány instrukce, které se nachází za měřeným blokem, čímž bude ovlivněn celkový výsledek měření, viz obrázek 4.2 [11, 15].



Obrázek 4.2: Vliv *OOE* na měření, kdy může být v měřeném bloku provedena instrukce nacházející se za měřeným blokem.

Tento nedostatek je možné odstranit voláním serializační instrukce *CPUID* za voláním instrukce *RDTSCP*, čímž bude garantováno, že budou změřeny jen instrukce nacházející se v měřeném bloku a zároveň nebude do měření započítán čas nutný k provedení instrukce *CPUID*, viz obrázek 4.3 [11, 15].



Obrázek 4.3: Příklad správného způsobu měření.

Konkrétní implementace funkcí pro měření času pomocí *TSC* je znázorněna ve výpisu 4.1. Tyto funkce jsou následně použity ve výpisu 4.2.

```
#include <inttypes.h>

static inline uint64_t start_cycle(void) {
        register unsigned a, d;

        __asm__ __volatile__ (
                "cpuid\n\t"
                "rdtsc\n\t"
        : "=a" (a),
          "=d" (d)
        :
        : "ebx", "ecx"
        );
        return (((uint64_t)d << 32) | a);
}

static inline uint64_t stop_cycle(void) {
        register unsigned high, low;

        __asm__ __volatile__ (
                "rdtscp\n\t"
                "movl %%eax, %[low]\n\t"
                "movl %%edx, %[high]\n\t"
                "cpuid\n\t"
        : [high] "=r" (high),
          [low] "=r" (low)
        :
        : "eax", "ebx", "ecx", "edx"
        );
        return (((uint64_t)high << 32) | low);
}
```

Výpis 4.1: Implementace funkcí pro měření času pomocí TSC v jazyce C/C++. Převzato z [11].

```c
#include <stdio.h>
#include <math.h>
#include <inttypes.h>

static void run_bench(double *const avg, double *const std_dev) {
        size_t i;

        *avg = 0.0;
        *std_dev = 0.0;
        for (i = 0; i < NUM_RUNS; i++) {
                uint64_t start, end;
                double t, measure;

                start = start_cycle();
                /* mereny kod */
                end = stop_cycle();

                measure = (double)(end - start);
                t = measure - *avg;
                *avg += t / (i + 1);
                *std_dev += (t * (measure - *avg));
        }
        *std_dev = sqrt(*std_dev / (NUM_RUNS - 1));
}
```

Výpis 4.2: Ukázka použití funkcí, které měří čas pomocí instrukcí RDTSC a RDTSCP. Převzato z [11].

Z předchozího textu vyplývá, že je měření času prostřednictvím *TSC* poměrně složitý proces, který s sebou přináší řadu problémů. Z toho důvodu vzikly speciální funkce, které se však liší v závislosti na použitém operačním systému. Na operačním systému Windows lze použít funkce *QueryPerformanceCounter* a *QueryPerformanceFrequency*, zatímco na operačním systému GNU/Linux je možné použít funkci *clock_gettime*.

4.1.2 Funkce QueryPerformanceCounter a QueryPerformanceFrequency

QueryPerformanceCounter (dále jen QPC) a *QueryPerformanceFrequency (dále jen QPF)* jsou funkce poskytované rozhraním *Win32 API*, nacházející se v hlavičkovém souboru *profileapi.h*. Tato dvojice funkcí představuje v podstatě rozhraní k hardwarovému zdroji času, čímž programátora odstíní od řešení problémů spojených například s měřením času pomocí *TSC*, a to včetně výběru vhodného zdroje času, který na daném zařízení poskytne nejrelevantnější výsledky, v dnešní době tedy typicky právě *TSC*. Funkce *QPC* slouží k získání hodinových cyklů, zatímco *QPF* slouží k získání frekvence použitého zdroje času, čímž je možné naměřenou hodnotu v hodinových cyklech převést na jednotku času [19].

Dříve, například na operačních systémech *Windows Vista* nebo *Windows Server 2008*, funkce *QPC* nevyužívala *TSC* jako základní zdroj času, ale využívala například *High Precision Event Timer*[2] *(HPET)*, případně *ACPI Power Management Timer*[3] *(PM timer)*. Tyto zdroje mají ovšem vyšší režijní náklady než *TSC* a navíc jsou sdíleny napříč všemi jádry, což limitovalo funkci, pokud byla volána z více jader. Jak již ale bylo zmíněno, dnešní *TSC* pracují s konstantní frekvencí, což vzali vývojáři *QPC* vpotaz, a proto jsou v novějších verzích *Windows* použity jako primární zdroj právě *TSC* registry, a to v případech, kdy je možné tyto registry sesynchronizovat při inicializaci systému [19].

4.1.3 Funkce clock_gettime

Funkce *clock_gettime* slouží k měření času na operačním systému *GNU/Linux*. Vstupními parametry této funkce jsou *ID* cílového zdroje času a ukazatel na strukturu pro uložení času. Pro měření běhu různých částí kódu je nejvhodnější uvést jako *ID* cílového zdroje času parametr *CLOCK_MONOTONIC*, případně *CLOCK_MONOTONIC_RAW*, a to z toho důvodu, že při dostupnosti invariantního *TSC* bude vnitřně využita přímo instrukce *rdtsc*. Při použití zmíněných parametrů má tedy tato funkce obdobné chování jako funkce *QPC*, s tím rozdílem, že není nutné převádět hodinové cykly na jednotku času, jelikož to funkce *clock_gettime* dělá automaticky [11].

[2]Zdroj času s vyššími režijními náklady než *TSC* [5, 10].
[3]Zdroj času s vyššími režijními náklady než *HPET* [5, 10].

4.1.4 Využití standardní knihovny

Standardní knihovny programovacích jazyků velice často poskytují funkce pro měření času. Mezi nabízenými funkcemi se často nachází funkce pro měření tzv. *High Resolutin Time Stamps*, tedy velice přesných časových vzorků. Tyto funkce často vnitřně využívají právě funkce *QueryPeformanceCounter* a *clock_gettime*, v závislosti na konrétním operačním systému. Výhodou daných funkcí je přenositelnost kódu, jelikož při přímém využití např. *QPC* není možné stejný kód využít i pro jiný operační systém než *Windows* a je tedy nutné použít podmíněnou kompilaci a nebo využít právě funkci ze standardní knihovny, která tuto práci udělá za programátora. U funkcí ze standardních knihoven je však nutné počítat s vyššími režijními náklady než u funkcí *QPC* a *clock_gettime*. Standardní knihovna programovacího jazyka Java poskytuje metodu *nanoTime*, která se nachází v balíčku *Java.lang*. Standardní knihovna C++ poskytuje metodu *now*, která se nachází v hlavičkovém souboru *chrono*, a to ve třídě *high_resolution_clock*.

4.2 Profilery

Výkonnostní profiler je nástroj, který slouží programátorovi k profilování vyvíjené aplikace. Oproti obyčejné manuální instrumentaci disponuje mnoha výhodami. Jednou z nich je obrovská úspora času, jelikož programátor nemusí manuálně vkládat kód pro měření výkonnostních parametrů a případně ho později mazat, ale pouze spustí profiler, který provede měření za něj. Mezi další z výhod se řadí poskytnutí mnohem větší škály výkonnostních parametrů, které mohou odhalit přesnou příčinu problematického chodu aplikace. Tyto parametry mohou být navíc zobrazeny graficky, což může značně ulehčit chápání naměřených hodnot.

Rozdělení metod, kterými profilery získávají data, není jednotné. Velká část zdrojů dělí tyto metody sběru dat na instrumentaci a vzorkování, přičemž definují instrumentaci jako metodu, během které jsou do profilované aplikace vkládány speciální instrukce a vzorkování jako neinvazivní metodu, během které není kód profilované aplikace nikterak modifikován a veškerá činnost spojena s profilingem probíhá vně aplikace, viz např. [2]. Toto rozdělení ovšem není přesné, jelikož vzorkování může být realizováno použitím instrumentace a ačkoliv by se tedy jednalo o vzorkování, byly by do profilovaného programu vloženy speciální instrukce, což nevyhovuje zmíněné definici vzorkování. Vhodnější proto může být dělit metody sběru dat na synchronní a asynchronní. Synchronní metody jsou ty, které získávají data pomocí speciálního kódu vloženého přímo do řídícího toku programu. Na

tomto principu funguje instrumentace, která se proto řadí právě mezi synchronní metody. Asynchronní metody jsou naopak ty, které nezasahují do řídícího toku programu a typicky fungují na principu obsluhy signálu, která byla vyvolána v důsledků výskytu nějaké události. V dnešní době využívají profilery nejčastěji dvě rozsáhlé metody – *instrumentaci* a *asynchronní vzorkování* [12, 16].

4.2.1 Asynchronní vzorkování

Jak již bylo zmíněno, tento typ vzorkování nezasahuje do řídícího toku programu a měření tedy probíhá vně aplikace [12, 16]. Mezi nejznámější typy *asynchronního vzorkování* patří *time-based vzorkování* [1, 17], *event-based vzorkování* a *instruction-based vzorkování* [1, 12, 16].

Time-based vzorkování je založeno na periodickém přerušování procesu, během kterého jsou získávána potřebná data, typicky hodnota v čítači instrukcí a zásobník volání. Periodické přerušování je realizováno nastavením časovače, který generuje přerušení v pravidelných intervalech. Při přerušení je volána obsluha přerušení, během které jsou získána potřebná data. Ze získaných dat je následně možné odhalit části kódu, ve kterých program trávil nejvíce času, jelikož právě tyto části jsou obsaženy v největším počtu naměřených vzorků, a to z toho důvodu, že je při přerušení programu prováděna s nejvyšší pravděpodobností instrukce nacházející se v časově náročné části kódu [1, 17].

Event-based vzorkování využívá vlastnosti přetečení hardwarových výkonnostních čítačů, při kterém je vyvoláno přerušení. Frekvence vzorkování je v tomto případě určena frekvencí přerušení vyvolaných přetečením čítačů. Vždy je tedy zvolen počet událostí x, po kterém dojde k přetečení čítače. Je-li maximální hodnota čítače rovna y, bude jeho hodnota nastavena na $y - x$, čímž dojde k přetečení po výskytu x událostí. Jak již bylo zmíněno, po přetečení dojde k přerušení, které musí být obslouženo. Během obsluhy je mimo jiné získána hodnota v čítači instrukcí, která reprezentuje právě vykonávanou instrukci. Tato instrukce je přiřazena k vyskytlé události, aby bylo možné zjistit, které instrukce mohou za výskyt daných událostí. Mezi přetečením a vlastní obsluhou vyvolaného přerušení je však prodleva, která způsobuje, že se v konrétních vzorcích nachází instrukce určující místo, ve kterém došlo k vyvolání obsluhy přerušení, nikoliv k přetečení čítače. Tyto dvě místa se navíc mohou lišit v řádu desítek instrukcí, jelikož dnešní procesory obsahují řadu nejrůznějších technik, například zpracování instrukcí mimo své pořadí, spekulativní provádění apod. Tento jev je označován jako *smyk* (angl. *skid*). Kvůli smyku je procesory značky *Intel* podporována me-

toda *Precise (Processor) Event-based sampling (PEBS)*, během které procesor ukládá instrukční ukazatel (spolu s dalšími informacemi) do předem určené vyrovnávací paměti – nedochází k přerušení a instrukční ukazatel obsahuje buďto přímo instrukci, během které došlo k přetečení čítače a nebo v nejhorším případě instrukci následující. Teprve po naplnění vyrovnávací paměti je vyvoláno přerušení, což snižuje režijní náklady, jelikož není přerušení voláno při každém přetečení čítače, ale až v případě, kdy je dostupné větší množství vzorků. Pouze některé události ale *PEBS* podporují, u těch ostatních je možné použít pouze *EBS* [12, 16].

Instruction-based vzorkování (IBS) je založeno na principu vzorkování instrukcí. Instrukce jsou periodicky vybírány ke sledování průběhu vykonávání instrukce v instrukční pipeline. Během zpracovávání a současného sledování instrukce jsou speciálním hardwarem získávány informace o výskytu významných událostí, například zda nedošlo k výpadku některé z mezipamětí, nebo zda byla instrukce úspěšně dokončena, či naopak zahozena. Jakmile je sledovaná instrukce dokončena, je vyvoláno přerušení a veškerá nasnímaná data jsou dostupná ke čtení [12, 16].

Dnešní *AMD* procesory využívají *IBS* pro řešení *smyku*, který se objevuje během *EBS*. Instrukční pipeline má dvě hlavní fáze – fázi načtení instrukce a fázi vykonání instrukc. Fáze načtení instrukce slouží k předávání dat dekodéru. Dekódované instrukce jsou následně provedeny během *fáze vykonání instrukce*. Jelikož jsou tyto dvě fáze oddělené, podporuje *IBS* dvě formy vzorkování – *fetch vzorkování* a *op vzorkování*. *IBS fetch vzorkování* poskytuje informace o fázi načtení instrukce, zatímco *IBS op vzorkování* poskytuje informace o fázi vykonání instrukce. Oba typy *IBS* využívají obdobné techniky vzorkování. *IBS* hardware periodicky vybírá operaci, která je označena a následně je monitorován průběh zpracování této operace. Události vyvolané monitorovanou operací jsou zaznamenány. Jakmile dojde k dokončení operace, je vyvoláno přerušení, během kterého jsou veškerá zaznamenaná data včetně instrukční adresy poskytnuta profileru, čímž dojde k přesnému přiřazení událostí k instrukci, která nasnímané události vyvolala [8, 12]. Během *IBS fetch vzorkování* je jednak získán počet provedených načtení a jednak dochází k periodickému výběru operace načtení – po definovaném počtu provedených načtení – která je tzv. *označena* (angl. *tagged*) a následně monitorována. Získány jsou tyto informace:

- načtená adresa,
- zda byla operace načtení dokončena či přerušena,
- zda byla načtená adresa v instrukční mezipaměti, respektive nebyla,

- zda došlo při převodu adresy k výpadku stránky v mezipaměti nazývané *Instruction Translation Lookaside Buffer*,
- velikost stránky,
- počet cyklů provedených mezi začátkem načtení a dokončením nebo přerušením načtení [8].

IBS op vzorkování počítá instrukční cykly a zároveň periodicky vybírá operaci – po definovaném počtu provedených instrukčních cyklů – která je označena a následně sledována. Vzorek je vytvořen pouze v případě, kdy dojde k dokočení označené operace. V opačném případě, kdy je operace zahozena, k vytvoření vzorku nedojde. Během *IBS op vzorkování* jsou získávány tyto informace:

- instrukční adresa,
- počet cyklů provedených mezi označením a odbavením operace,
- počet cyklů provedených mezi dokončením a odbavením operace,
- informace o predikci větvění,
- informace o tom, zda se nejedná o resync operaci, která způsobuje zpoždění pipeline,
- zda došlo ke čtení z paměti, či uložení do paměti, včetně bližších informací – například zda nedošlo k výpadku vyrovnávací paměti, virtuální a fyzická adresa požadované paměti apod [8].

4.2.2 Instrumentace

Instrumentace je metoda, při níž je do cílového, měřeného programu vkládán speciální kód (tzv. *sondy*), který slouží k měření různých výkonnostních parametrů (událostí). Tyto události se dělí na *atomické* a *intervalové*. Atomickou událost je možné změřit jedním měřením, kdy jsou v čase výskytu dané události získána potřebná data. Typickou atomickou událostí je například alokace paměti o určité velikosti. Na druhé straně intervalovou událost je možné získat změřením dvou atomických událostí na *začátku* a *konci* měřeného bloku, typicky se tedy jedná například o měření časů potřebných k vykonání cyklů, funkcí apod [12, 16].

Instrumentace může poskytnout jak přesná data, například při počítání provedených alokací paměti, tak i data nepřesná. Vykonání měřícího kódu totiž trvá určitou dobu, pokud by tedy bylo nutné změřit například časy

potřebné k vykonání všech funkcí, bylo by nutné vložit sondy na začátek a konec každé funkce. Funkce může být ale volána v těle jiné funkce, což znamená, že může být do celkového času zanesen čas nutný k vykonání měřícího kódu, což do měření vnáší chybu. Kromě toho dnešní procesory poskytují řadu optimalizací – např. branch prediction. Kvůli těmto optimalizacím jsou procesory závislé na pořadí vykonávaných instrukcí. Je-li tedy měřená funkce příliš krátká, může být po přidání měřící instrukce na začátek a konec daného bloku narušen způsob, jakým by byla za normálních okolností, tedy bez měřících instrukcí, funkce v procesoru provedena. Pokud je takováto funkce v programu volána mnohokrát, nebude profiler schopen poskytnout přesné časové srovnání mezi touto krátkou funkcí a funkcemi, které jsou delší. V důsledku to může znamenat, že bude programátor optimalizovat funkci, která ve skutečnosti, za normálního chodu, nemá vliv na pomalý chod aplikace [2].

Instrumentaci lze realizovat několika způsoby. Prvním způsobem je *instrumentace zdrojového kódu*, kdy je měřící kód vkládán přímo do kódu zdrojového, nejčastěji manuálně programátorem. Výhodou je v takovém případě možnost určit přesné části kódu, které mají být měřeny. Nevýhodou je naopak časová náročnost a možnost zanesení chyb do zdrojového kódu. Z toho důvodu dokáží některé profilery provést automatickou instrumentaci zdrojového kódu – například vložit sondy na začátek a konec každé funkce. Mezi instrumentaci zdrojového kódu se řadí také tzv. *library wrapping* metoda, kdy jsou knihovní funkce nahrazovány instrumentovanými verzemi. Druhou možností je využít k instrumentaci *kompilátor*, kdy je kompilátoru pomocí parametru sděleno, že má provést instrumentaci kódu. V takovém případě je ovšem nutno počítat s vyššími režijními náklady než u instrumentace zdrojového kódu. Další z možností je *instrumentace binárního souboru*, kdy profiler využije některý z nástrojů pro přepisování binárního souboru, například *Dyninst API* nebo *MAQAO*. Výhodou takového řešení je nezávislost na zdrojovém kódu, který například nemusí být dostupný a navíc není nutný opětovný překlad kódu. Takovéto nástroje navíc často dokáží provést také *dynamickou instrumentaci* neboli instrumentaci běžícího programu. Pro interpretované jazyky je možné využít také *interpretovanou instrumentaci*, při níž profiler spolupracuje přímo s virtuálním strojem nebo interpretem za běhu aplikace [12, 16].

5 VTune Amplifier

Intel VTune Amplifier je nástroj vyvíjený společností *Intel Corporation*, vysoce optimalizovaný k analýze aplikací běžících na procesorech značky Intel. Tento nástroj využívá k analýze aplikací dva typy sběru dat. Prvním typem je tzv. *User-Mode Sampling and Tracing Collection*, což je v podstatě vzorkování založené čistě na čase a je možné jej použít při hledání časově náročných míst v kódu. Druhým typem je tzv. *Hardware Event-based Sampling Collection*, což je vzorkování využívající hardwarovou jednotku *PMU*. VTune poskytuje několik předdefinovaných analýz, například:

1. Hotspots – hledání časově nejnáročnějších míst v programu,
2. Microarchitecture Exploration – analýza využití dostupných prostředků postkytovaných daným hardwarem,
3. Memory Access – analýza přístupů k paměti (mezipaměti a hlavní paměti),
4. HPC Performance Characterization – analýza výkonu aplikace a využití matematického koprocesoru s informacemi o provedené vektorizaci, použité *Single Instruction Multiple Data (SIMD)* instrukční sadě apod.

Kromě předdefinovaných analýz je možné vytvářet vlastní analýzy, ve kterých je možné nastavit frekvenci vzorkování, případně limitovat režim výkonnostních čítačů jen na počítání, vybrat seznam měřených událostí apod. Při výběru měřených událostí je ovšem nezbytně nutné daným událostem rozumět a vědět, které události jsou na daném procesoru dostupné. U předdefinovaných analýz je obrovskou výhodou, že *VTune* kombinuje několik událostí dohromady – vytváří tzv. *metriky*, které často uživateli poskytnou mnohem detailnější informace o analyzované aplikaci.

5.1 Microarchitecture Exploration

V této analýze je použito několik metrik založených na percentuálním využití pipeline slotů a hodinových cyklů. Pipeline slot v tomto případě reprezentuje hardwarové prostředky nutné k provedení jedné mikrooperace[1] (*uOp*).

[1]Nejzákladnější operace proveditelná procesorem. Instrukce je typicky množinou několika mikrooperací.

Počet pipeline slotů, nazývaný též jako *šířka zřetězeného zpracování* (angl. *pipeline width*), určuje počet mikrooperací alokovatelných a odbavitelných během jednoho hodinového cyklu. Šířka zřetězeného zpracování je dnes typicky rovna čtyřem. Zřetězené zpracování je během analýzy děleno na *Front End* a *Back End*. *Front End* reprezentuje tu část zřetězeného zpracování, ve které dochází k načtení (výběru) instrukce z instrukční mezipaměti nebo paměti a dekódování instrukce na jednu či vícero mikrooperací. Proces „doručení" mikrooperace z *Front End* části do *Back End* části je nazýván jako *alokace*, která v podstatě reprezentuje přidělení *uOp* některému z dostupných *pipeline slotů*. Jakmile je v *Back End* části *uOp* dokončena, dojde k tzv. *odbavení uOp*, kdy je operace „odstraněna" z *pipeline slotu*. Jakmile jsou veškeré mikrooperace konkrétní instrukce odbaveny, dojde k odbavení celé instrukce ze zřetězeného zpracování a tedy jej opustí. Pokud při zřetězeném zpracování nedojde k alokaci *uOp* během hodinovém cyklu, jedná se o tzv. *zpoždění pipeline slotu*[2]. Nemůže-li *Front End* část předat *uOp Back End* části, ačkoliv by ji tato část mohla přijmout, jedná se o tzv. *Front End Bound* slot, jelikož byl tento slot zpožděn kvůli *Front End* části. Naopak je-li *uOp* připravena k „doručení" *Back End* části, která ovšem nemá volný *pipeline slot* a nemůže tedy *uOp* přijmout, jedná se o tzv. *Back End Bound* slot, jelikož byl slot zpožděn kvůli *Back End* části. Jsou-li důvodem zpoždění *Back End* i *Front End*, je slot klasifikován jako *Back End Bound* [11, 13].

Dojde-li k *alokaci uOp*, existují dva případy, které mohou nastat. V prvním případě dojde k odebrání *uOp* z důvodu *chybné predikce větvení* (angl. *branch mispredict*), kdy je *pipeline slot* zařazen do kategorie *Bad Speculation*. Ve druhém případě je *uOp* odbavena a pipeline slot umístěn do kategorie *Retiring*. Zatímco v kategoriích *Front End Bound*, *Back End Bound* a *Bad Speculation* je žádoucí mít zařazen co nejnižší počet slotů, do kategorie *Retiring* by mělo být zařazeno slotů co možná nejvíce, jelikož tato kategorie signalizuje jejich efektivní využití. Tyto kategorie dále obsahují podkategorie, které poskytují další, detailnější informace o průběhu vykonání měřeného programu. Například kategorie *Back End Bound* se dále dělí na *Core Bound* a *Memory Bound*. Tyto subkategorie už ovšem nemusí být měřeny v percentuálním využití pipeline slotů, nýbrž v percentuálním využití hodinových cyklů. Zvolená jednotka závisí na konkrétní *PMU*, resp. událostech, které je *PMU* schopna měřit [11, 13]. Jelikož se subkategorie opět dále dělí, vzniká poměrně složitá hierarchie metrik, mezi které se řadí například:

- *Core Bound (COREB)* – procento pipeline slotů zpožděných kvůli procesoru,

[2]stav, kdy *pipeline slot* neobržel novou *uOp* ke zpracování

- *Memory Bound (MEMB)* – procento pipeline slotů zpožděných z paměťových důvodů,
- *L1 Bound (L1B)* – procento hodinových cyklů, během kterých nedošlo k výpadku mezipaměti L1, ale došlo ke zpoždění,
- *L2 Bound (L2B)* – procento hodinových cyklů, během kterých došlo ke zpoždění, kdy nebyla nalezena data v L1, ale až v L2,
- *L3 Bound (L3B)* – procento hodinových cyklů, během kterých došlo ke zpoždění, kdy nebyla nalezena data v L2, ale až v L3,
- *DRAM Bound (DRAMB)* – procento hodinových cyklů, během kterých došlo ke zpoždění, kdy nebyla data nalezena v žádné mezipaměti a musela být získána až z hlavní paměti

5.2 HPC Performance Characterization

Tato analýza používá především metriky, které určují podíl specifických operací (např. v plovoucí řadové čárce) vůči všem operacím. Je tedy například možné zjistit, kolik procent operací bylo v plovoucí řadové čárce, kolik procent operací v pohyblivé řadové čárce bylo vektorizováno, nebo které SIMD instrukční sady byly použity.

6 Testovací úlohy

Tato kapitola obsahuje seznam vybraných testovacích úloh, které byly použity pro analýzu výkonu vybraných překladačů. Zároveň jsou zde zmíněny důvody výběru těchto úloh – tedy jaký aspekt práce překladače měly jednotlivé úlohy otestovat. Každá úloha poskytuje přepínač *-i*, pomocí kterého je možné nastavit počet iterací měřené metody.

6.1 Úloha č. 1

První zvolenou úlohou bylo násobení matic pomocí klasického, přímočarého řešení, kdy je prvek v i-tém řádku a j-tém sloupci vypočítán jako skalární součin vektoru i-tého řádku první matice s vektorem j-tého sloupce druhé matice. Jedná se o velice neefektivní řešení z hlediska využití dostupných cache pamětí. Pomocí této úlohy mělo být zjištěno, zda dokáže některý z vybraných překladačů provést vhodné transformace cyklů, které povedou k lepšímu využití *L1–L3* cache pamětí.

Pro uložení matic byla využita jednorozměrná pole, čímž bylo zajištěno, že se budou prvky jednotlivých matic nacházet v jednom souvislém paměťovém bloku. V jazyce *C++* byl sice pro uložení matic využit kontejner *std::vector*, vždy ale byly získány ukazatele na paměť používanou daným kontejnerem, a to pomocí metody *data()*, díky čemuž nemusel překladač provádět *inlining* metod. Kromě toho bylo vhodné pracovat s ukazately i kvůli *auto vektorizaci* kódu, která je v takovém případě překladačem snáze proveditelná.

Matice byly v této úloze načteny ze souborů, aby překladač nemohl předvypočítávat některé výsledky. Zároveň byla v této úloze měřena pouze doba běhu metody, která prováděla daný výpočet, což znamená, že do výsledné doby běhu nebyly započítány časy nutné k alokaci pamětí, načtení matic apod.

6.2 Úloha č. 2

Druhou zvolenou úlohou bylo taktéž násobení matic, nicméně se značně efektivnejším řešením. V této úloze totiž byly využity dvě techniky používané ke snížení počtu výpadků cache pamětí, a to *transpozice* druhé matice a tzv. *loop tiling*. Především díky *loop tilingu* bylo násobení matic převedeno z

paměťově náročné úlohy na úlohu výpočetně náročnou.

Matice byly v této úloze uloženy dvěma způsoby. Jednak byla opět využita jednorozměrná pole a jednak byly použity také generické kontejnery ve zdrojových kódech napsaných v jazycích C++, Java a Object Pascal. V jazyce C++ byl využit kontejner *std::vector*, v Javě *ArrayList* z balíčku *java.util* a v Object Pascalu *TVector* z jednotky *GVector*. Obdobně jako v předchozí úloze, byly i v této úloze matice načteny ze souborů, aby nedocházelo k předvypočítávání výsledků. Měřena byla pouze metoda, která prováděla násobení matic.

Cílem této úlohy bylo otestovat, do jaké míry zvládnou jednotlivé překladače optimalizovat kód, který obsahuje velké množství operací v plovoucí řadové čárce a také zjistit, zda dojde s využitím kontejnerů k poklesu výkonu a nebo zda dokáží překladače kontejnery optimalizovat (např. pomocí *inliningu*), aby byl pokles výkonu jen minimální.

6.3 Úloha č. 3

Třetí zvolenou úlohou byla *Gaussova eliminační metoda*, která obsahuje poměrně velké množství operací v plovoucí řadové čárce a zároveň obsahuje většinu základních aritmetických operací, a to výpočet absolutní hodnoty, dále sčítání, odčítání, násobení a dělení.

Soustava rovnic byla v této úloze načtena ze souboru a uložena do dvou jednorozměrných polí reprezentujících pravou a levou stranu soustavy. Pro uložení čísel v plovoucí řadové čárce byly využity dva datové typy – *double* a *float*. Díky tomu bylo možné porovnat vliv zvoleného datového typu na rychlost doby běhu výsledného kódu a přesnost výsledků dle použité úrovně optimalizace.

Úloha byla spuštěna na datech, pro která byl zjištěn přesný výsledek, který byl následně porovnáván s výsledky vypočítanými pomocí překladači vygenerovaných kódů.

6.4 Úloha č. 4

Poslední testovací úlohou, která byla v této práci měřena, bylo tzv. *zpětné vyhledávání* (angl. *backtracking*), což je úloha těžce optimalizovatelná překladačem, jelikož se nejedná o tzv. *koncovou rekurzi* (angl. *tail recursion*), kterou by se překladač mohl pokusit převést na iterativní formu. V této úloze byl hledán počet cest v grafu z vrcholu a do vrcholu b, délky maximálně n, kde a, b a n jsou čísla definovaná uživatelem při spuštění. Cílem

této úlohy bylo zjistit, do jaké míry zvládnou překladače takovýto typ úlohy optimalizovat, jelikož se jedná o úlohu se složitějšími strukturami a velkým množstvím funkčních volání, které by nemělo být možné pomocí *inliningu* „odstínit“.

7 Testovací procedura

Tato kapitola obsahuje veškeré informace o testovacím prostředí, použitých kompilátorech, zvolených měřících metodách a způsobu měření.

7.1 Testovací prostředí

Veškeré testy byly provedeny na notebooku Dell Inspiron 15 (7559) s operační pamětí DDR3 o kapacitě 8192 MB, čtyřjádrovým procesorem Intel Core i7-6700HQ (2,6 GHz, 3,5 GHz Turbo Boost) a operačním systémem Windows 10 Home (10.0, build 18363).

Kvůli zajištění co možná nejvyšší přesnosti a stability výsledků byly vypnuty veškeré aplikace a služby, které neměly přímý vliv na chod systému Windows a které mohly nějakým způsobem ovlivnit výsledky, tedy například antivirus. Zároveň byly zakázány veškeré síťové adaptéry. Kromě toho byla vypnuta přímo v Biosu technologie Speedstep, čímž byla automaticky vypnuta také technologie TurboBoost, díky čemuž pracoval procesor testovacího zařízení s téměř konstantní frekvenci *2,58–2,59* GHz.

7.2 Zvolené kompilátory a optimalizace

Vybrány byly některé ze známých a často používaných kompilátorů jazyků *C*, *C++*, *Java* a *Object Pascal*. Úrovně optimalizací byly vybrány všechny, které daný překladač nabízí, tedy jak *bezpečné* úrovně, tak i úrovně *nebezpečné*. Mezi *nebezpečné* úrovně se řadí úrovně využívající optimalizace, které mohou ovlivnit přesnost výsledku, tedy například úroveň *Ofast*, poskytovaná překladačem *GCC*, která využivá optimalizaci, resp. soubor optimalizací *-ffast-math*. Při specifikaci takové úrovně může překladač pracovat v *nebezpečném režimu*, ve kterém lze s čísly v plovoucí řadové čárce pracovat tak, jako by pro ně platila např. asociativita, která ovšem dle standardu *IEEE-754* neplatí. Díky tomu může překladač různě „přeskládat“ jednotlivé operace, což má vliv na přesnost výsledků. Jelikož žádný z použitých překladačů neposkytuje úroveň optimalizace, která by využila nejvyšší dostupnou *SIMD* instrukční sadu daného zařízení, byla vytvořena vlastní úroveň, dále označována jako *Ov*, s jejíž specifikací může překladač danou sadu použít.

Pro jazyky C a C++ byly zvoleny překladače:

- MinGW-w64 (64 bit, 8.1.0) s úrovněmi optimalizace *O0*, *O1*, *O2*, *O3*, *Os*, *Ofast*, *Ov (O3 + ffast-math + mavx2)*.
- Visual C++ (64 bit, 19.16.27034) s úrovněmi optimalizace *Od*, *O1*, *O2*, *Ov (O2 + fp:fast + arch:AVX2)*.
- Intel C++ (64 bit, 19.0.0.117, 20180804) s úrovněmi optimalizace *Od*, *O1*, *O2*, *O3*, *Os*, *Ov (O3 + fp:fast=2 + arch:CORE-AVX2 + Qprec-div- + Qipo-)*.

Pro jazyk Object Pascal byl zvolen volně dostupný překladač:

- Free Pascal (64 bit, 3.0.4) s úrovněmi optimalizace *O0 (O-)*, *O1*, *O2*, *O3*, *O4*, *Os*, *Ov (O4 + Oofastmath + CfAVX2)*.

Javovské zdrojové soubory, narozdíl od Object Pascalu a C/C++, nejsou překládány na kód strojový, ale jsou přeloženy do mezikódu nazývaného *bytecode*, který je následně zpracováván virtuálním strojem *Java Virtual Machine (JVM)*, který slouží k interpretaci tohoto mezikódu. Kromě toho ale *JVM* obsahuje také *Just-in-time (JIT)* překladač, který překládá, na základě analýz, často používané části zmíněného mezikódu na kód strojový, přičemž provádí také většinu optimalizací. Z toho důvodu byly vybrány k následné analýze právě virtuální stroje, konkrétně:

- HotSpot (jdk-11.0.7+10),
- OpenJ9 (jdk-11.0.6+10).

7.3 Zvolené měřící metody

Tato podkapitola obsahuje seznam vybraných metod pro měření výkonnostních parametrů, včetně zdůvodnění výběru konkrétních metod.

7.3.1 Doba běhu

Pro měření doby běhu bylo možné zvolit manuální instrumentaci s využitím *TSC*, a nebo použít profiler, v tomto případě VTune Amplifier. Z toho důvodu bylo provedeno několik testů, na základě kterých měla být vybrána vhodná metoda měření času. Pro manuální instrumentaci byla využita v jazycích C, C++ a Object Pascal funkce *QPC*. Jelikož nebylo možné využít tuto funkci v Javě, byla v javovských zdrojových souborem použita metoda *nanoTime*, ze třídy *System*, která vnitřně využívá právě *QPC*. Nejprve bylo nutné otestovat, zda je na na dané platformě využíván funkcí *QPC* relevatní

zdroj času, v tomto případě *TSC*. V C++ proto byla vytvořena funkce pro měření času pomocí instrukce *RDTSC*. Ta byla následně využita spolu s funkcí *QPC* pro změření bloku kódu – výsledky následně potvrdily, že byl zdroj času *TSC* funkcí *QPC* využit, jelikož vyšly totožné výsledky. Posléze bylo nutné porovnat manuální instrumentaci s profilerem – došlo tedy ke změření stejného bloku kódu oběma metodami. Výsledky měření se nachází v tabulce 7.1.

Tabulka 7.1: Porovnání metod pro měření času, při různém zatížení procesoru.

	Zatížení CPU ~0 %		Zatížení CPU ~15 %	
Měřící metoda	Počet měření	Průmerná hodnota (ms)	Počet měření	Průmerná hodnota (ms)
VTune	50	61,8	50	70,3
QPC	50	61,7	50	68,2

Jelikož byly časy naměřené oběma metodami téměř stejné, byla nakonec vybrána pro měření času manuální instrumentace, a to z toho důvodu, že je možné časy získané touto metodou postupně ukládat například do pole a následně z nich vypočítat přímo v programu statistické veličiny – např. kvartily, směrodatnou odchylku, variační koeficient apod. Profiler byl tedy využit až při podrobnější analýze kódu.

7.3.2 Doba překladu

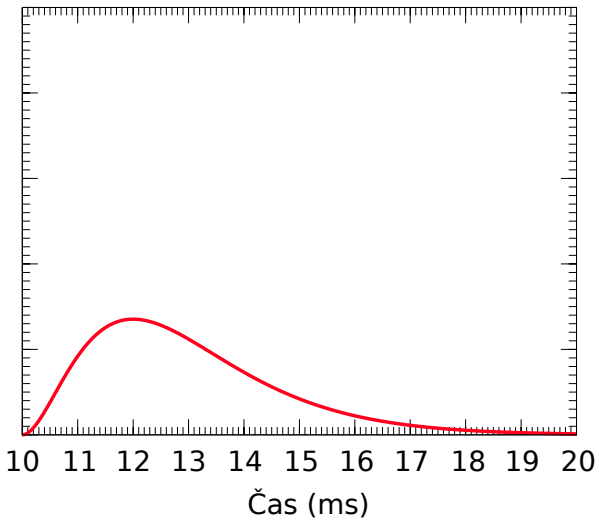
Pro měření doby překladu byla použita jednoduchá aplikace *Tim*, napsaná v programovacím jazyce C++, volně dostupná na *githubu*, viz [9]. Ta slouží k měření času na operačním systému *Windows*. Tato aplikace měří čas pomocí funkcí *GetTickCount64* a *QPC* – je tedy na uživateli, který výstup zvolí. V tomto případě byl k časovému srovnání zvolen výstup z funkce *QPC*, který je přesnější.

7.3.3 Paměťová náročnost

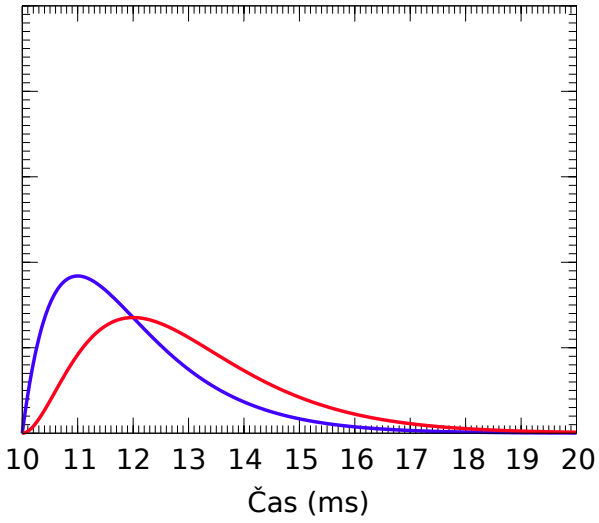
Pro měření paměťové náročnosti byla použita opět aplikace *Tim*, která kromě měření času měří také paměťovou náročnost měřené aplikace.

7.4 Způsob měření

Nejprve byl pro každou úroveň optimalizace změřen čas, nutný k vykonání výsledného kódu. Každý kód byl změřen celkem čtyřikrát, s x iteracemi během jednoho měření – tedy celkem bylo naměřeno $4x$ různých časů. Z každého měření byly vypočítány statistické veličiny – minimální a maximální hodnota, průměrná hodnota, kvartily, směrodatná odchylka a variační koeficient. Při měření byl během každé iterace získán čas $t = t_1 + t_2 + ... + t_n$, kde t_1 představuje čas nutný k vykonání měřeného kódu a t_2 až t_n jsou chyby zanesené do měření z různých důvodů – například kvůli přepínání kontextu operačním systémem. Z toho důvodu se samozřejmě naměřené hodnoty více či méně lišily a bylo proto nutné zvolit vhodnou statistickou veličinu, která by výsledná měření co nejpřesněji charakterizovala, aby bylo možné kompilátory objektivně porovnat. Jelikož mají často aplikace – především ty výpočetně náročné – pravostranně asymetrické rozdělení, viz obrázek 7.1, mohlo by být vhodné zvolit jako referenční čas minimální hodnotu daného měření, která by teoreticky měla být nejméně vzdálena od času t_1, který je hledán. Na obrázku 7.2 je však vidět nevýhoda minimální hodnoty, jelikož ačkoliv mají dvě různá měření, dvou různých kódů stejnou minimální hodnotu, je dle obrázku 7.2 zřejmé, že je kód reprezentovaný modrou křivkou rychlejší, ačkoliv by byly oba výsledné kódy prohlášeny za stejně rychlé. Další možností proto bylo zvolit za referenční čas průměrnou hodnotu. Ta ovšem může být značně ovlivněna extrémními výkyvy v měření, a proto byl nakonec zvolen k porovnávání jednotlivých měření medián. Jak již bylo zmíněno, pro každou úroveň optimalizace byl výsledný kód změřen čtyřikrát. Následně byly porovnány mediány těchto čtyř měření, přičemž bylo měření s nejnižším mediánem zvoleno za referenční pro danou úroveň. Následně mohly být z těchto měření vybrány tři kódy k následné analýze, a to *nejrychlejší kód bez optimalizací*, *nejrychlejší kód s bezpečnou úrovní optimalizace* a *nejrychlejší kód s nebezpečnou úrovní optimalizace*. Poté již byla provedena analýza vybraných kódů, a to pomocí profileru *VTune Amplifier*. Na závěr byla změřena doba překladu a paměťová náročnost těchto kódů.



Obrázek 7.1: Ukázka pravostranně asymetrického rozdělení typického pro měření časů nutných k vykonání výpočetně náročných aplikací.



Obrázek 7.2: Ukázka dvou pravostranně asymetrických rozdělení se stejnou minimální hodnotou.

8 Výsledky měření a jejich analýza

Tato kapitola obsahuje porovnání výsledných kódů, které byly vygenerovány vybranými překladači s různými úrovněmi optimalizace. Dále tato kapitola obsahuje hlubší analýzu dosažených výsledků.

8.1 Úloha č. 1

Na základě naměřených dat, která se nachází v příloze A, byly vybrány úrovně optimalizace, pomocí kterých byl překladači vygenerován nejrychlejší kód. Seznam těchto úrovní byl zanesen do tabulky 8.1.

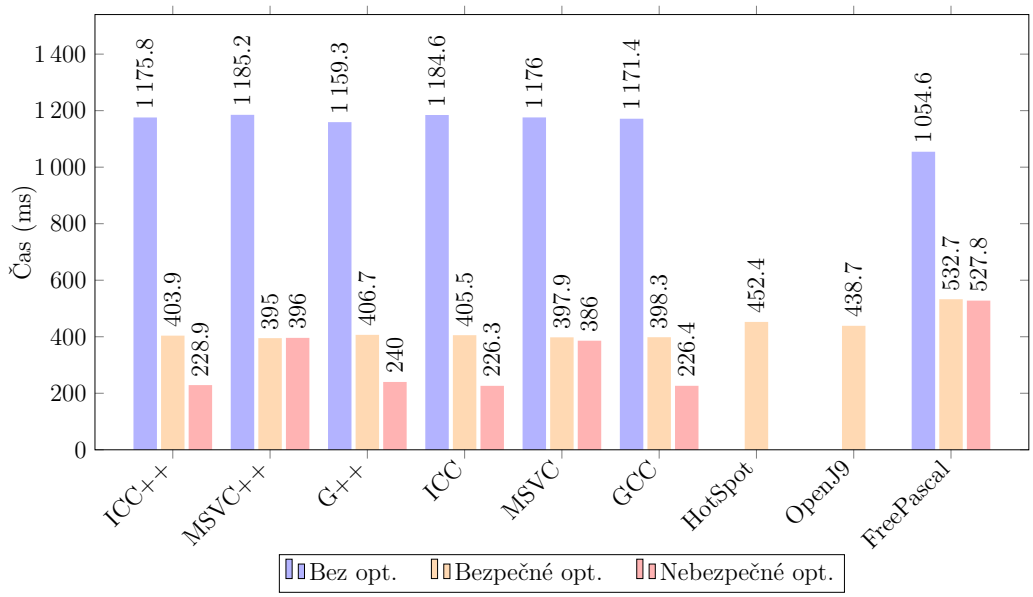
Tabulka 8.1: Seznam úrovní optimalizace, se kterými byl v úloze č. 1 vytvořen vybranými překladači nejrychlejší kód.

Jazyk	Překladač	Úroveň optimalizace	
		Bezpečná	Nebezpečná
C++	ICC	O1	Ov
	MSVC	O2	Ov
	GCC	O3	Ov
C	ICC	O1	Ov
	MSVC	O2	Ov
	GCC	O2	Ov
Object Pascal	FreePascal	O3	O4

8.1.1 Doba běhu

Z přílohy A byly vybrány doby běhu výsledných kódů vygenerovaných překladači s vybranými úrovněmi optimalizace. Srovnání jednotlivých časů se nachází na obrázku 8.1.

Kód bez optimalizací Nejrychlejší neoptimalizovaný kód byl vygenerován překladačem *Free Pascal*. Tento překladač zvládnul vygenerovat kód s nejnižším počtem odbavených instrukcí a zároveň nejvyšším procentem operací v plovoucí řadové čárce.

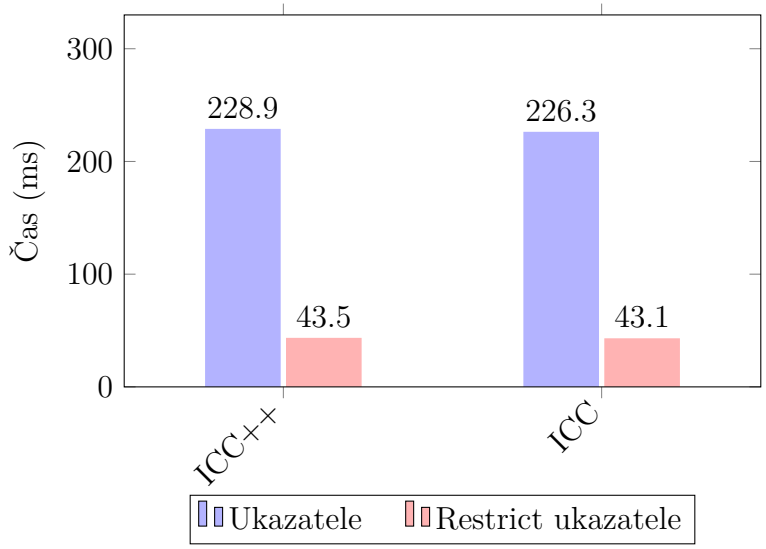


Obrázek 8.1: Srovnání dob běhu překladači vygenerovaných kódů v úloze č. 1, a to v závislosti na zvoleném typu optimalizace.

Kód s bezpečnými optimalizacemi Nejrychlejší kód s *bezpečnými optimalizacemi* byl vygenerován *C/C++* překladači, ze kterých byl nejvýkonnějším překladač *MSVC*, který zvádnul vygenerovat kód s naprosto nejvyšším procentem operací v plovoucí řadové čárce, které dokonce přesáhlo hranici 50 %. Nicméně i přesto, že obsahoval kód vygenerovaný tímto překladačem vysoké procento operací v pohyblivé řadové čárce, byl ve srovnání s ostatními výslednými kódy překladačů *C/C++* rychlejší jen o pár procent, jelikož nedošlo ke snížení výpadků cache pamětí a vzrostlo tedy procento zpožděných *pipeline slotů* – metrika *MEMB* byla až o 15 % vyšší než u kódu generovaného překladačem *ICC*.

Kód s nebezpečnými optimalizacemi Z výsledných kódů, při jejichž generování použily překladače nebezpečné optimalizace, byl nejrychlejší kód vytvořen opět *C/C++* překladači, a to konrétně překladači *ICC* a *GCC*. Třetí z těchto překladačů, tedy překladač *MSVC*, značně zaostával za oběma překladači, jelikož neprovedl vektorizaci kódu, kterou překladače *ICC* a *GCC* provedly. S použitím vektorizace došlo k tzv. zabalení všech operací v plovoucí řadové čárce – operandy byly zabaleny do vektorů. Ačkoliv bylo překladačům explicitně sděleno, aby použily *SIMD* instrukční sadu *AVX2*, která pracuje s *256-bitovými* vektory, byly veškeré operandy v plovoucí řadové čárce zabaleny pouze do vektorů poloviční délky – použit byl speciální režim *AVX-128*, který je určen pro práci se *128-bitovými* vektory. Z obou zmíně-

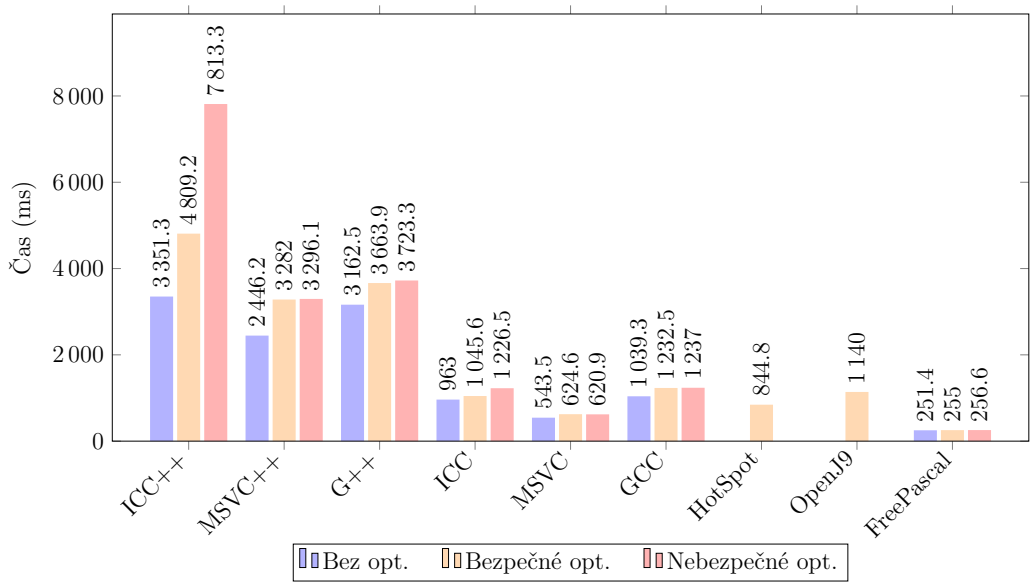
ných překladačů byl vygenerován nepatrně rychlejší kód překladačem *ICC*, který využil kromě samotné *SIMD* instrukční sady *AVX* také rozšiřující sadu *FMA*, což byl pravděpodobně jeden z důvodů rychlejšího běhu kódu. Žádný z překladačů neprovedl agresivní transformace cyklů, mezi které lze zařadit například optimalizace *loop interchange*, *loop tiling*, *unroll-and-jam* apod. Z toho důvodu bylo k maticím přistupováno stále velice neefektivním způsobem, což v konečném důsledku znamenalo, že nedošlo ke snížení počtu výpadků *L1–L3* cache pamětí. Byly ovšem provedeny další testy, pomocí kterých bylo zjištěno, že překladač *ICC* zvládne jako jediný provést agresivní transformace cyklů, pokud jsou mu poskytnuty informace o nezávislosti paměťových bloků, ve kterých jsou uloženy matice. Toho lze docílit použitím tzv. *restrict* ukazatelů, pomocí kterých je překladači explicitně sděleno, že se paměťové bloky nepřekrývají, respektive že se jedná o jediný ukazatel, pomocí kterého je možné k dané paměti přistoupit. Díky tomu může daný překladač provést další, agresivnější optimalizace. V tomto konkrétním případě byly překladačem provedeny optimalizace *loop interchange* a *unroll-and-jam*, díky kterým došlo k mnohem efektivnějšímu využití cache pamětí, jelikož byl počet jejich výpadků snížen o více jak 99 %. Zároveň mohly být veškeré operandy v plovoucí řadové čárce zabaleny do *256-bitových* vektorů, čímž došlo k obrovskému nárůstu výkonu, viz obrázek 8.2.



Obrázek 8.2: Vliv použitých restrict ukazatelů v úloze č. 1 na dobu běhu výsledného kódu, který byl vygenerován překladačem *ICC* s úrovní optimalizace *Ov*.

8.1.2 Doba překladu

Statistické veličiny, které byly vypočteny z naměřených časů nutných ke kompilaci zdrojových kódů s vybranými úrovněmi optimalizace, se nachází v příloze B. Referenční hodnoty (mediány) jsou graficky znázorněny na obrázku 8.3.



Obrázek 8.3: Srovnání časů nutných k překladu zdrojových souborů realizujících úlohu č. 1, a to v závislosti na použitém překladači a úrovni optimalizace.

Nejrychlejší překlad byl proveden překladačem *Free Pascal*. Ten zvládnul přeložit kód několikanásobně rychleji než ostatní překladače. Zároveň nedošlo k téměř žádnému nárůstu doby překladu ani při použití různých úrovní optimalizace, přičemž je ovšem nutno podotknout, že nebyly provedené optimalizace příliš efektivní (ve srovnání s C/C++ překladači). Nejdéle trval překlad *C++* zdrojových kódů, a to především překladačem *ICC*, který sice vygeneroval nejrychlejší kód s použitím nebezpečných optimalizací, ale dle výsledků je zřejmé, že to mělo obrovský dopad na dobu překladu, která byla přibližně dvakrát delší než u ostatních *C/C++* překladačů.

8.1.3 Paměťová náročnost

V tabulce 8.2 se nachází paměťová náročnost jednotlivých výsledných kódů. Paměťově nejméně náročný kód byl vygenerován překladači *C/C++*. Výsledný kód generovaný překladačem *Free Pascal* potřeboval o *0,5–1* MB paměti více. Mnohonásobně více paměti bylo použito virtuálními stroji *Javy*,

které si zarezervovaly paměť, jež v podstatě vůbec nepotřebovaly. Z toho důvodu byly výsledné kódy *Javy* dvacetkrát až čtyřicetkrát náročnější na paměť než ostatní kódy.

Tabulka 8.2: Paměťová náročnost výsledných kódů realizujících úlohu č. 1, v závislosti na použitém překladači a úrovni optimalizace.

Jazyk	Překladač	Použitá paměť (MB)		
		Bez optimalizace	Bezpečná úroveň opt.	Nebezpečná úroveň opt.
C++	ICC	10,54	10,54	10,55
	MSVC	10,52	10,52	10,52
	GCC	11,03	11,04	11,04
C	ICC	10,49	10,47	10,47
	MSVC	10,48	10,47	10,47
	GCC	10,54	10,54	10,54
Java	HotSpot	–	222,90	–
	OpenJ9	–	436,92	–
Object Pascal	FreePascal	11,48	11,48	11,48

8.2 Úloha č. 2

Tato podkapitola obsahuje výsledky měření úlohy č. 2. Na základě naměřených dat, která se nachází v příloze C, byly vybrány nejrychlejší úrovně optimalizace, jejichž výčet byl zanesen do tabulky 8.3.

Tabulka 8.3: Seznam úrovní optimalizace, se kterými byl v úloze č. 2 vytvořen vybranými překladači nejrychlejší kód.

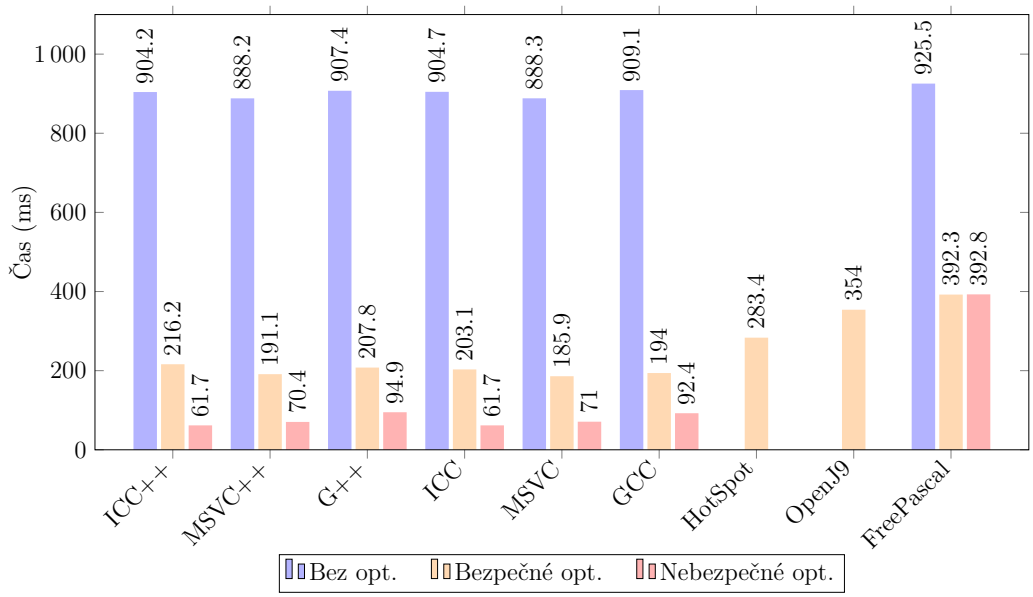
Jazyk	Překladač	Úroveň optimalizace	
		Bezpečná	Nebezpečná
C++	ICC	O1	Ov
	MSVC	O2	Ov
	G++	O3	Ov
C	ICC	O1	Ov
	MSVC	O2	Ov
	GCC	O2	Ov
Object Pascal	FreePascal	O3	O4

8.2.1 Doba běhu vyprodukovaného kódu

Z přílohy C byly vybrány doby běhu (mediány) výsledných kódů vytvořených překladači s vybranými úrovněmi optimalizace. Porovnání jednotlivých časů se nachází na obrázku 8.4.

Kód bez optimalizací Nejrychlejší kód bez optimalizací byl vygenerován překladači prog. jazyků *C/C++*, ze kterých byl nejvýkonnější překladač *MSVC*, který vytvořil kód s nejvyšším procentem operací v plovoucí řadové čárce.

Kód s bezpečnými optimalizacemi Nejefektivnější kód s bezpečnými optimalizacemi byl vygenerován *C/C++* překladači. Tyto překladače vygenerovaly kódy, které byly mnohem efektivnější než kódy generované Java a Object Pascal překladači. V tabulce 8.4 je vidět srovnání, ze kterého plyne, že byl *C/C++* překladači vytvořen kód s nejvyšším procentem operací v plovoucí řadové čárce a zároveň nejnižším počtem odbavených instrukcí. Mimoto využily všechny tři překladače tzv. *Conditional Move (CMOV)* instrukce, což mohlo být také jedním z faktorů rychlejšího běhu výsledných kódů. Ze všech tří *C/C++* překladačů vykazoval nejlepší výsledky překladač *MSVC*, který zvládnul vygenerovat kód s nejvyšším procentem operací



Obrázek 8.4: Srovnání dob běhu překladači vygenerovaných kódů v úloze č. 2, a to v závislosti na zvoleném typu optimalizace.

v plovoucí řadové čárce, což vzhledem k povaze úlohy znamenalo, že byl kód na daném zařízení vykonán nejrychleji – ačkoliv samozřejmě není procento operací v plovoucí řadové čárce jediným měřítkem efektivity kódu, jelikož zde hrajou roli další faktory, například vektorizace kódu. Právě ta byla provedena překladačem G++, což je také důvod, proč byl tímto překladačem vygenerován kód se značně nižším procentem operací v pohyblivé řadové čárce než ostatními dvěma *C/C++* překladači. Provedená vektorizace však nebyla příliš efektivní, jelikož bylo vektorizováno pouze 33 % operací v plovoucí řadové čárce, a proto byl výsledný kód pomalejší než kódy generované překladači *ICC* a *MSVC*.

Kód s nebezpečnými optimalizacemi Ačkoliv je překladačem *Free Pascal* poskytována úroveň optimalizace *O4*, s jejíž specifikací může dojít k provedení nebezpečných optimalizací, nebyl s využitím této úrovně kód nikterak zrychlen. Dokonce ani s použitím vlastní úrovně optimalizace *Ov* ke zrychlení kódu nedošlo – nebyla provedena ani byť jen částečná vektorizace kódu. Naopak překladače *C/C++* zvládly s vlastní úrovní optimalizace *Ov* kód značně zrychlit, a to nejméně o 50 %. Došlo k rozbalení smyček (angl. *loop unrolling*) a následné vektorizaci kódu, přičemž byla k vektorizaci využita všemi třemi překladači *SIMD* instrukční sada *AVX* a došlo tedy k zabalení operandů do *256-bitových* vektorů. Ze všech tří *C/C++* překladačů vykazoval nejlepší výsledky překladač *ICC*, který jednak vygeneroval kód s nejvyšším

procentem operací v plovoucí řadové čárce (viz tabulka 8.4) a jednak použil jako jediný rozšiřující instrukční sadu *FMA*, stejně jako v předchozí úloze.

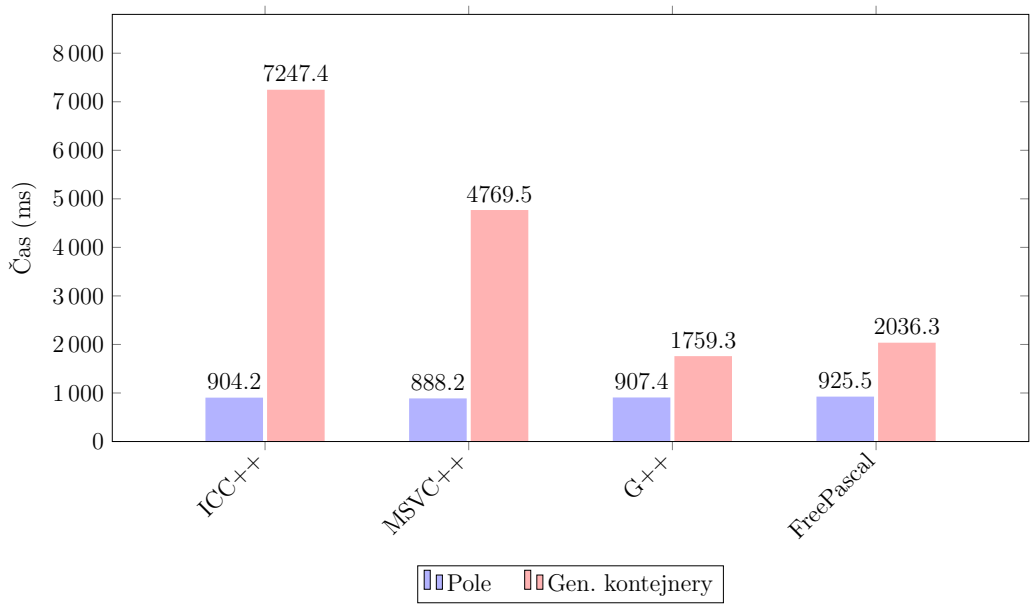
Tabulka 8.4: Procento operací v plovoucí řadové čárce a počet odbavených instrukcí během vykonávání výsledných kódů realizujících úlohu č. 2, a to v závislosti na zvoleném překladači a typu optimalizace.

Jazyk	Překladač	Bezpečná optimalizace		Nebezpečná optimalizace	
		% FP ops	Odbavené instrukce	% FP ops	Odbavené instrukce
C++	ICC	36,0	1724548800	55,2	192493600
	MSVC	39,0	1463415200	37,9	320829600
	G++	24,9	1667255200	31,0	382844800
C	ICC	36,4	1722947200	50,2	192972000
	MSVC	40,3	1465453600	34,2	322337600
	GCC	30,9	1688866400	32,6	383000800
Java	HotSpot	17,1	3042634400	–	–
	OpenJ9	15,1	3756084800	–	–
OP	FreePascal	13,3	4273599200	14,7	3772111200

Kód s použitím generických kontejnerů V jazycích *C++*, *Java* a *Object Pascal* byly v této úloze využity pro uložení matic kromě polí také generické kontejnery. Naměřené časy nutné k vykonání výsledných kódů využívajících generické kontejnery se nachází na konci přílohy C. Tyto časy byly následně porovnány s časy potřebnými k vykonání kódů využívajících klasická pole.

Na obrázku 8.5 je vidět srovnání neoptimalizovaných verzí těchto kódů. Na základě dat uvedených na tomto obrázku lze konstatovat, že došlo s použitím generických kontejnerů ke značnému snížení výkonu neoptimalizovaných verzí výsledných kódů. Důvodem je obrovské množství funkčních volání, která značně navyšují dobu běhu. Nejvyšší nárůst času byl zaznamenán u kódu generovaného překladačem *ICC*, kdy byl výsledný kód přibližně osmkrát pomalejší.

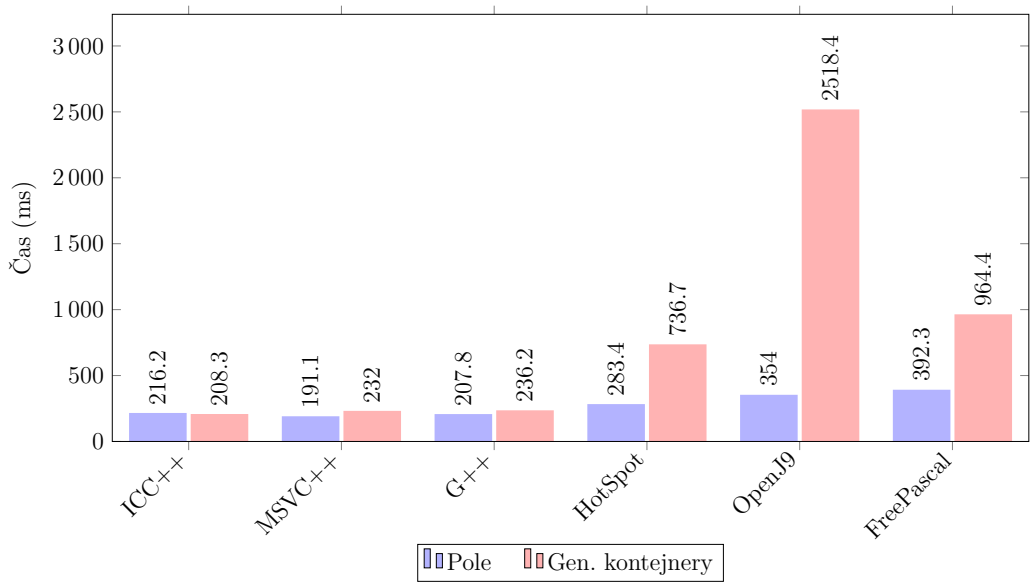
Obrázek 8.6 obsahuje porovnání kódů s bezpečnými optimalizacemi. Jak je vidět, u *C/C++* překladačů nedošlo k žádnému (*ICC*) a nebo jen malému (*MSVC* a *G++*) snížení výkonu, a to díky provedení tzv. *inliningu*. Naopak u *Java* a *Object Pascal* překladačů došlo ke značnému nárůstu doby běhu výsledných kódů. Zároveň je vidět, že se značně liší doby běhu v závislosti na použitém virtuálním stroji *Javy*, a to z toho důvodu, že zatímco virtuální stroj *HotSpot* (resp. jeho *JIT* kompilátor) provedl *inlining* metod třídy *ArrayList*, vrituální stroj *OpenJ9* vůbec neprovedl *inlining* metody *get()*, ze



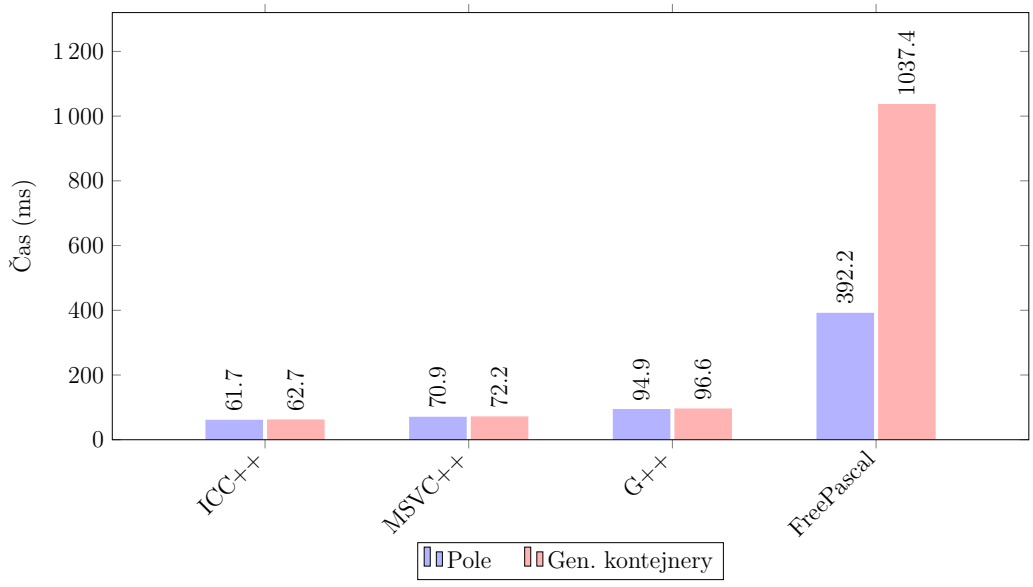
Obrázek 8.5: Porovnání dob běhu neoptimalizovaných verzí kódů v úloze č. 2, a to v závislosti na použité datové struktuře pro uložení matic.

třídy *ArrayList*, která je v kódu hojně využívána. Kromě toho vzrostl počet výpadků cache pamětí *L1–L3*, jelikož *ArrayList* nepodporuje primitivní datové typy, a proto musí být pro každou hodnotu vytvořena instance třídy *Double*, obalující danou hodnotu. V *ArrayListu* jsou ovšem uloženy pouze reference, nikoliv hodnoty, což znamená, že se dané hodnoty pravděpodobně nenachází v jednom souvislém paměťovém bloku, čímž dochází ke značnému zvýšení počtu výpadků cache pamětí.

Na obrázku 8.7 je vidět porovnání kódů s *nebezpečnými optimalizacemi*. U *C/C++* překladačů došlo jen k velice malému nárůstu doby běhu (přibližně o 1 až 2 %), jelikož byl opět proveden *inlining* metod. Naopak kód s generickými kontejnery, přeložený překladačem *Free Pascal*, byl téměř třikrát pomalejší. Nedošlo totiž k *inliningu* funkcí, a to i přesto, že použitý kontejner *inline* funkce obsahoval. Následně bylo zjištěno, že překladač *Free Pascal* neprovádí *inlining* funkcí, pokud není kód přeložen s přepínačem *-Si*. Došlo tedy k opětovnému překladu kódu, tentokrát již s tímto přepínačem. Výsledky ovšem vyšly naprosto stejné, jelikož ani s přepínačem *-Si* daný překladač *inlining* funkcí neprovedl.



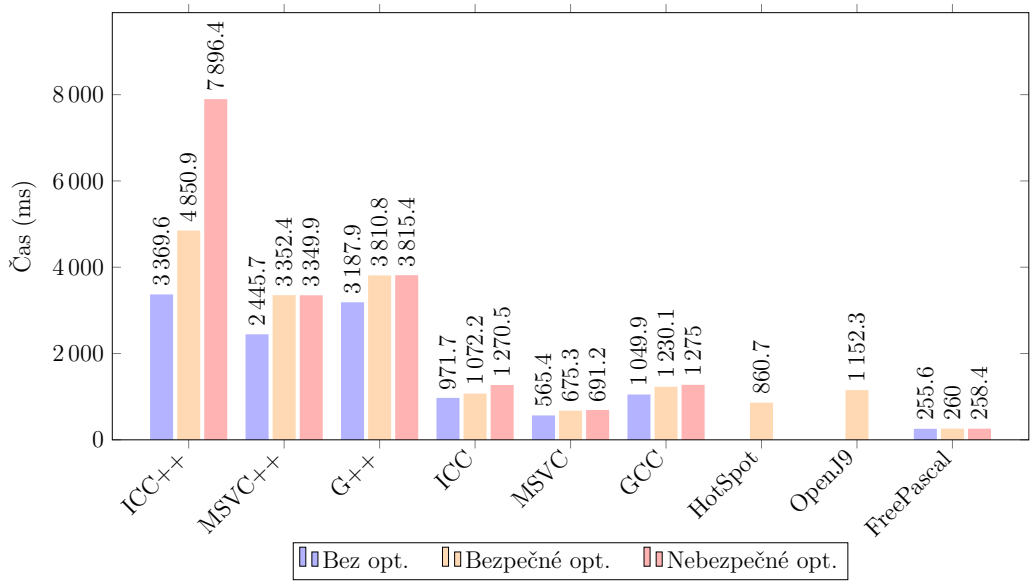
Obrázek 8.6: Porovnání dob běhu kódů s bezpečnými optimalizacemi v úloze č. 2, a to v závislosti na použité datové struktuře pro uložení matic.



Obrázek 8.7: Porovnání dob běhu kódů s bezpečnými optimalizacemi v úloze č. 2, a to v závislosti na použité datové struktuře pro uložení matic.

8.2.2 Doba překladu

Pro vybrané úrovně optimalizace byly změřeny časy nutné k překladu kódu s příslušnými úrovněmi. Změřené časy, resp. statistické veličiny vypočtené z těchto časů, se nachází v příloze D. Grafické srovnání těchto časů se nachází na obrázku 8.8.



Obrázek 8.8: Srovnání časů nutných k překladu zdrojových souborů realizujících úlohu č. 2, a to v závislosti na použitém překladači a úrovni optimalizace.

Nejrychleji byl kód přeložen překladačem *Free Pascal*, a to ve všech třech kategoriích. Nejpomaleji byly přeloženy *C++* zdrojové kódy, jejichž překlad trval třikrát až sedmkrát déle než překlad zdrojových kódu napsaných v jazyce *C*. Z *C/C++* překladačů byl kód přeložen nejrychleji překladačem *MSVC*. Naopak nejdelší dobu trval překlad kompilátorem *ICC*, který sice provedl nejefektivnější vektorizaci kódu, ale za cenu značného zpomalení překladu.

8.2.3 Paměťová náročnost

V tabulce 8.5 se nachází paměťová náročnost jednotlivých výsledných kódů. Výsledky byly velice podobné výsledkům z předchozí úlohy. Paměťově nejnáročnější kód byl opět kód provedený virutálním strojem *OpenJ9*, který si zarezervoval mnohonásobně více paměti, než bylo nutné. Naopak paměťové nejméně náročné byly kódy vygenerované překladači *C/C++*.

Tabulka 8.5: Paměťová náročnost výsledných kódů realizujících úlohu č. 2, v závislosti na použitém překladači a úrovni optimalizace.

Jazyk	Překladač	Použitá paměť (MB)		
		Bez optimalizace	Bezpečná úroveň opt.	Nebezpečná úroveň opt.
C++	ICC	10,54	10,54	10,55
	MSVC	10,52	10,51	10,52
	GCC	11,08	11,08	11,08
C	ICC	10,47	10,47	10,47
	MSVC	10,48	10,47	10,47
	GCC	10,53	10,54	10,54
Java	HotSpot	–	222,02	–
	OpenJ9	–	436,53	–
Object Pascal	FreePascal	11,51	11,51	11,51

8.2.4 Poznámka

V této úloze byly pro *loop tiling* použity dvě velikosti bloku. Pro kódy s bezpečnými optimalizacemi byla velikost bloku rovna 16. Pro nebezpečné optimalizace byl použit blok o velikosti 1024, a to z toho důvodu, že s nebezpečnými optimalizacemi často dochází k transformacím cyklů, které jsou efektivnější.

8.3 Úloha č. 3

Na základě naměřených časů, které se nachází v příloze E, byly vybrány úrovně, se kterými vygenerovaly překladače v této úloze nejrychlejší kód. Přehled vybraných úrovní se nachází v tabulce 8.6.

Tabulka 8.6: Seznam úrovní optimalizace, se kterými byl v úloze č. 3 vytvořen vybranými překladači nejrychlejší kód.

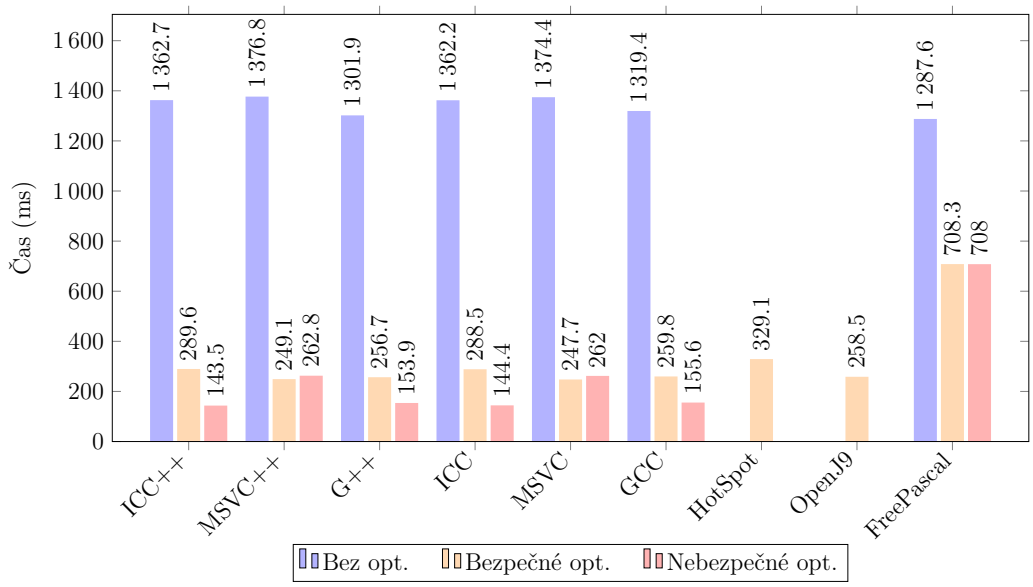
Jazyk	Překladač	Úroveň optimalizace	
		Bezpečná	Nebezpečná
C++	ICC	O1	Ov
	MSVC	O2	Ov
	GCC	O3	Ov
C	ICC	O1	Ov
	MSVC	O2	Ov
	GCC	O3	Ov
Object Pascal	FreePascal	O3	O4

8.3.1 Doba běhu

Z naměřených časů, které se nachází v příloze E, byly vybrány doby běhu výsledných kódů s vybranými úrovněmi optimalizace. Dané časy jsou zobrazeny na obrázku 8.9.

Kód bez optimalizací Nejefektivnější kód bez optimalizací, z hlediska doby běhu, byl dle naměřených dat vygenerován překladačem *Free Pascal*. Nepatrně delší dobu běhu měl kód vygenerovaný překladačem *GCC*. Ačkoliv bylo během vykonávání kódu, který byl vytvořen překladačem *GCC*, zpožděno nižší procento *pipeline slotů*, obsahoval tento kód také nižší procento operací v plovoucí řadové čárce, což v konečném důsledku znamenalo, že byl kód pomalejší přibližně o 1 %.

Kód s bezpečnými optimalizacemi Nejrychlejší kód s bezpečnými optimalizacemi byl vygenerován překladači *C/C++*. Z těch byl naprosto nejrychlejší kód překladače *MSVC*, který obsahoval nejvyšší procento operací v plovoucí řadové čárce a zároveň byl během vykonání tohoto kódu odbaven nízký počet instrukcí (druhý nejnižší), viz tabulka 8.7. Stejně jako v



Obrázek 8.9: Srovnání dob běhu překladači vygenerovaných kódů v úloze č. 3, a to v závislosti na zvoleném typu optimalizace.

předchozí úloze byla překladačem *GCC* provedena vektorizace kódu, kterou žádný jiný překladač v *bezpečném módu* neprovedl. Ačkoliv byly zabaleny téměř všechny operace (resp. operandy) v plovoucí řadové čárce do vektorů délky *128-bitů*, nebyl kód stále tak efektivní jako kód generovaný překladačem *MSVC*.

Kód s nebezpečnými optimalizacemi Překladač *ICC* zvládnul v *nebezpečném módu* vygenerovat kód s nejkratší dobou běhu, stejně jako v předchozích úlohách. Výsledný kód obsahoval vysoké procento operací v plovoucí řadové čárce a k jeho provedení bylo nutné odbavit nejnižší počet instrukcí, viz tabulka 8.7. Zároveň byly tímto překladačem využity kromě *SIMD* instrukční sady *AVX* také sady *AVX2* a *FMA*. Naopak překladač *MSVC*, stejně jako v úloze č. 1, neprovedl vektorizaci kódu, čímž byl výsledný kód značně pomalejší než kódy generované ostatními *C/C++* překladači. Vektorizaci neprovedl ani překladač *Free Pascal*, jehož kód byl nejpomalejší.

Kód s využitím datového typu float V příloze E se nachází také doby běhu výsledných kódů, ve kterých byl k uložení čísel v plovoucí řadové čárce využit *32-bitový* datový typ *float* (resp. *Single* v jazyce *Object Pascal*). Na obrázku 8.10 je vidět porování dob běhu neoptimalizovaných verzí výsledných kódů, a to v závislosti na použitém datovém typu. Z dat uvedených na tomto obrázku vyplývá, že s použitím datového typu *float* nedošlo k příliš

Tabulka 8.7: Procento operací v plovoucí řadové čárce a počet odbavených instrukcí během vykonávání výsledných kódů realizujících úlohu č. 3, a to v závislosti na zvoleném překladači a typu optimalizace.

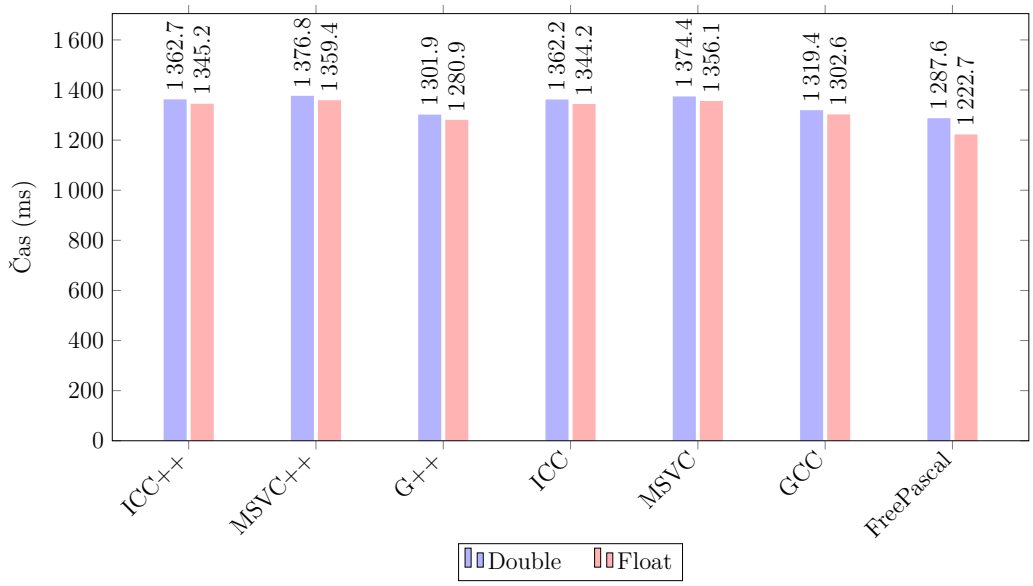
Jazyk	Překladač	Bezpečná optimalizace		Nebezpečná optimalizace	
		% FP ops	Odbavené instrukce	% FP ops	Odbavené instrukce
C++	ICC	28,2	2692573000	41,5	356889000
	MSVC	37,2	1924832000	42,7	1616914000
	G++	21,5	1863810000	25,3	603785000
C	ICC	28,3	2692196000	42,7	357396000
	MSVC	37,0	1951209000	46,8	1609179000
	GCC	18,6	1864460000	23,3	610818000
Java	HotSpot	23,2	2859142000	–	–
	OpenJ9	23,9	3165591000	–	–
OP	FreePascal	9,5	7382843000	10,0	7382674000

razantnímu snížení doby běhu, která klesla pouze o 1–5 %. Znatelnější nárůst výkonu se objevil u výsledných kódů s *bezpečnými optimalizacemi*, přičemž byl nejvyšší nárůst výkonu zaznamenán u kódu vygenerovaného překladačem *GCC*. Jak již bylo zmíněno, překladač *GCC* provedl v bezpečném módu jako jediný vektorizaci kódu, která sice s použitím datového typu *double* nebyla příliš efektivní, ale s datovým typem *float* došlo ke značnému snížení doby běhu (až o 46 %), viz obrázek 8.11. Důvodem je, že s použitím datového typu *float* mohl být do vektorů zabalen dvojnásobný počet operandů – v tomto případě čtyři namísto dvou. U výsledných kódů s nebezpečnými optimalizacemi došlo po použití datového typu *float* dokonce ke snížení doby běhu o 59–63 %, a to u překladačů, které provedly vektorizaci kódu – tedy *ICC* a *GCC*, viz obrázek 8.12.

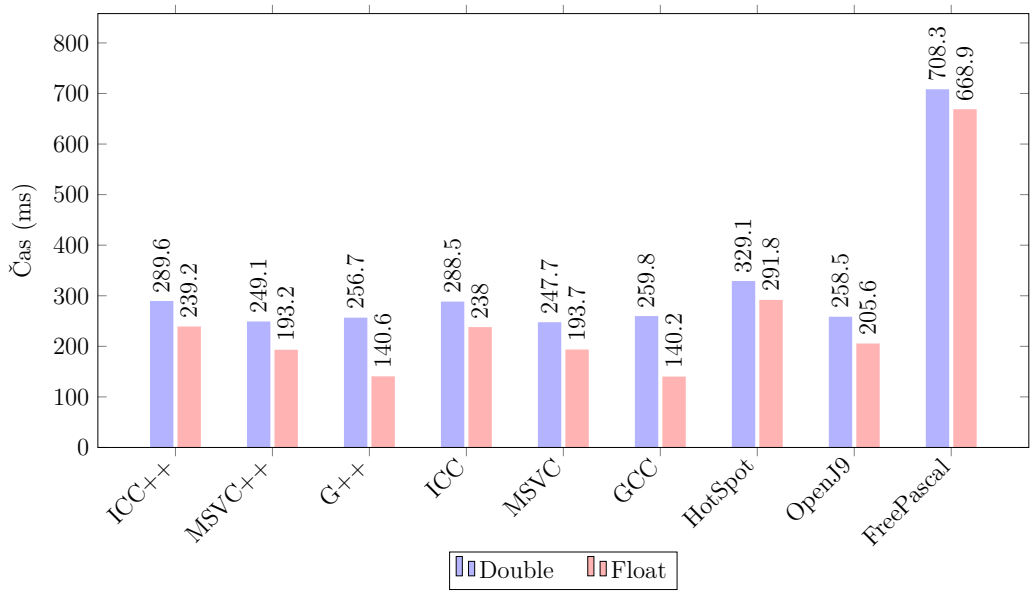
8.3.2 Přesnost výsledků

Pomocí této úlohy byla porovnána také přesnost výsledků v závislosti na zvoleném datovém typu a úrovni optimalizace. Testovací data byla navržena tak, aby byly všechny neznámé dané soustavy rovny stejnému číslu, a to konkrétně číslu $0.0\overline{9920634}$.

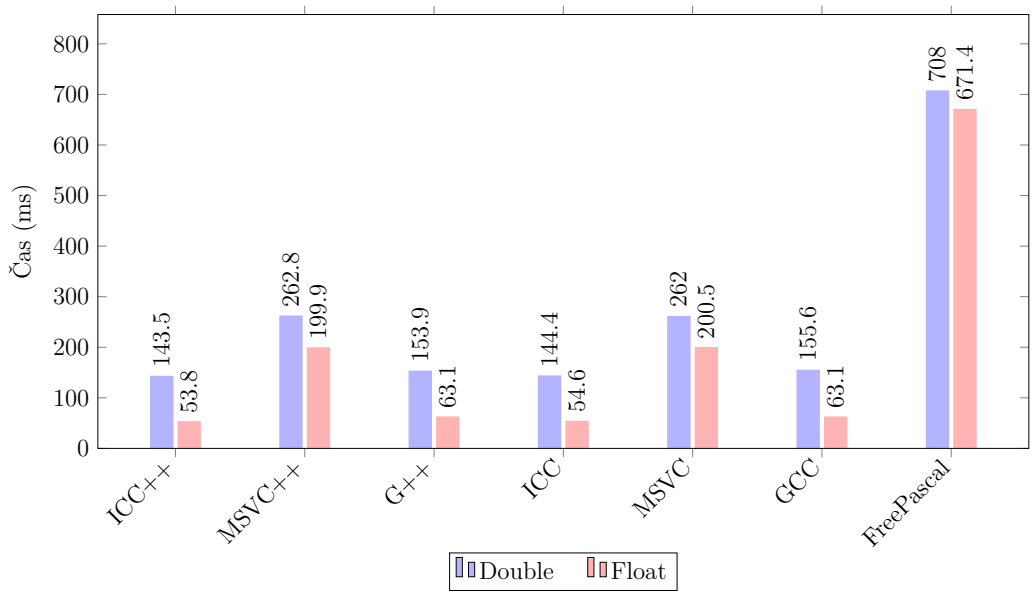
Datový typ double Veškeré výsledné kódy bez optimalizací a také kódy s bezpečnými optimalizacemi vypočítaly totožné výsledky (z hlediska uložení čísel v paměti). Ačkoliv byly výsledky vypočítané pomocí Javovských výsledných kódů mírně odlišné, šlo pouze o jinou metodu zaokrouhlování



Obrázek 8.10: Srovnání dob běhu výsledných kódů (bez optimalizací) realizujících úlohu č. 3, a to v závislosti na zvoleném datovém typu.



Obrázek 8.11: Srovnání dob běhu výsledných kódů (s bezpečnými optimalizacemi) realizujících úlohu č. 3, a to v závislosti na zvoleném datovém typu.



Obrázek 8.12: Srovnání dob běhu výsledných kódů (s nebezpečnými optimalizacemi) realizujících úlohu č. 3, a to v závislosti na zvoleném datovém typu.

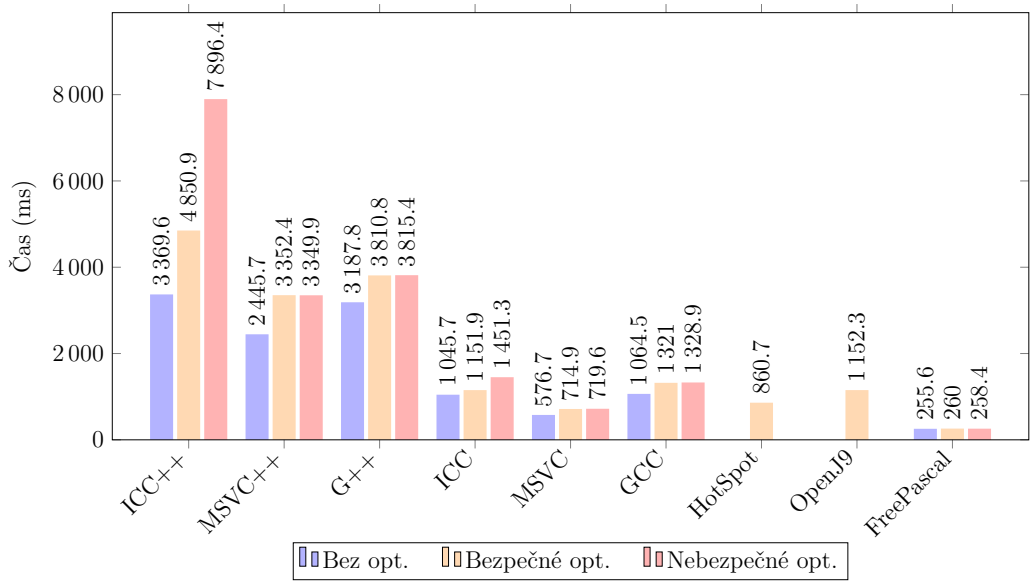
čísel při převodu na řetězec. Například zatímco pomocí *javovského* kódu byla neznáma x_{21} rovna číslu 0.09920634920631, pomocí kódu generovaného *C/C++* překladačem byla tato neznámá rovna číslu 0.099206349206309996. V počítači jsou ovšem obě tato čísla reprezentována naprosto stejně a jsou si tedy rovna. Veškeré výsledky byly totožné s přesným výsledkem až do dvanáctého desetinného místa (včetně). Od třináctého desetinného místa se vypočtené hodnoty různě lišily, jelikož pro čísla v plovoucí řadové čárce neplatí asociativita a zároveň nejsou v počítači tato čísla uložena exaktně, protože jsou aproximována.

S použitím nebezpečných úrovní optimalizace došlo u výsledných kódů s provedenou vektorizací ke zpřesnění výsledků. Ty byly přesnější o jedno desetinné místo – výsledky tedy byly při srovnání s přesným výsledkem rozdílné až na čtrnáctém desetinném místě. Důvodem pravděpodobně bylo použití některých *SIMD* instrukcí, které mohou být nejen rychlejší, ale také přesnější.

Datový typ float S použitím datového typu *float* došlo ke značnému snížení přesnosti výsledků. Získané hodnoty byly rovny přesné hodnotě pouze do třetího desetinného místa (včetně). Od čvrtého desetinného místa se již hodnoty značně lišily a ani vektorizované kódy přesnost výsledků nezvýšily.

8.3.3 Doba překladu

Pro vybrané úrovně optimalizace byly v této úloze změřeny časy nutné k překladu kódů s příslušnými úrovněmi. Statistické veličiny vypočtené ze změřených časů se nachází v příloze F. Dané hodnoty jsou graficky znázorněny na obrázku 8.13.



Obrázek 8.13: Srovnání časů nutných k překladu zdrojových souborů realizujících úlohu č. 3, a to v závislosti na použitém překladači a úrovni optimalizace.

Nejrychlejší překlad byl v této úloze proveden překladačem *Free Pascal*, obdobně jako v obou předchozích úlohách. Nejpomaleji byly přeloženy zdrojové soubory napsané v programovacím jazyce *C++*, přičemž naprosto nejpomalejší překlad byl vykonán překladačem *ICC*.

8.3.4 Paměťová náročnost

V tabulce 8.8 se nachází paměťová náročnost jednotlivých výsledných kódů. Paměťově nejnáročnější kód byl opět kód provedený virutálním strojem *OpenJ9*, který si zarezervoval mnohonásobně více paměti, než bylo nutné. Naopak paměťové nejméně náročné byly v této úloze kódy vygenerované překladači *C/C++*.

Tabulka 8.8: Paměťová náročnost výsledných kódů realizujících úlohu č. 3, v závislosti na použitém překladači a úrovni optimalizace.

Jazyk	Překladač	Použitá paměť (MB)		
		Bez optimalizace	Bezpečná úroveň opt.	Nebezpečná úroveň opt.
C++	ICC	16,54	16,54	16,54
	MSVC	16,51	16,51	16,51
	GCC	17,05	17,05	17,05
C	ICC	16,51	16,51	16,51
	MSVC	16,50	16,50	16,50
	GCC	16,56	16,56	16,56
Java	HotSpot	–	179,82	–
	OpenJ9	–	436,94	–
Object Pascal	FreePascal	17,67	17,67	17,67

8.4 Úloha č. 4

Na základě naměřených dat, která se nachází v příloze G, byly vybrány úrovně optimalizace, pomocí kterých byl překladači vygenerován nejrychlejší kód. Seznam těchto úrovní byl zanesen do tabulky 8.9.

Tabulka 8.9: Seznam úrovní optimalizace, se kterými byl v úloze č. 4 vytvořen vybranými překladači nejrychlejší kód.

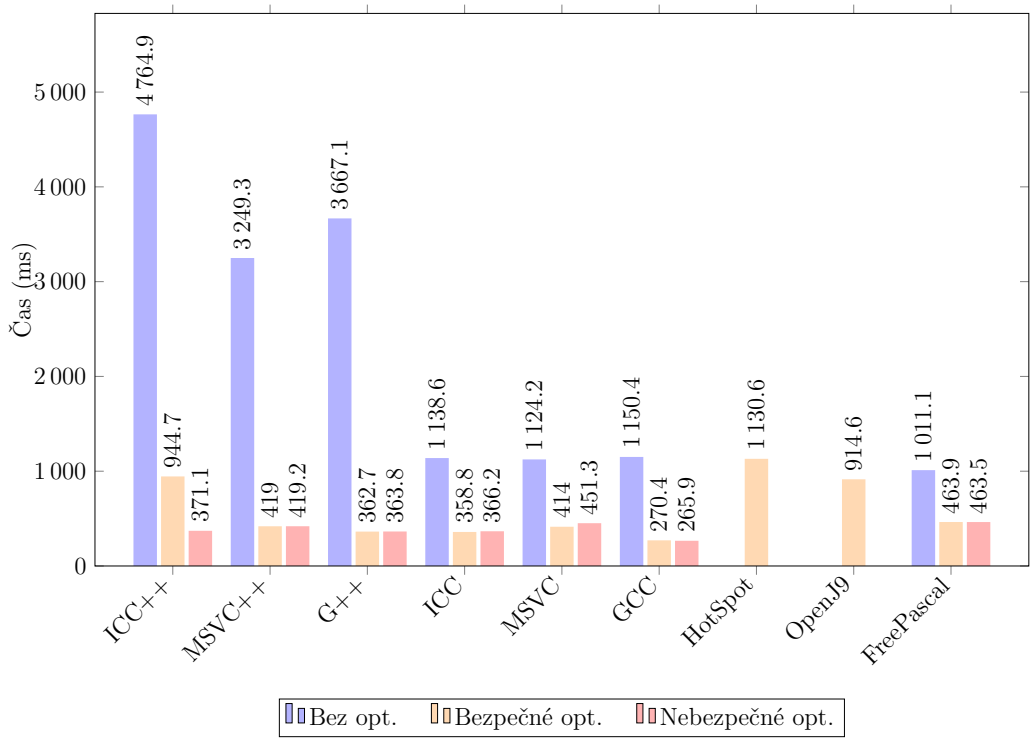
Jazyk	Překladač	Úroveň optimalizace	
		Bezpečná	Nebezpečná
C++	ICC	O1	O3
	MSVC	O2	Ov
	GCC	O2	Ov
C	ICC	O1	Ov
	MSVC	O2	Ov
	GCC	O3	Ov
Object Pascal	FreePascal	O2	Ov

8.4.1 Doba běhu

Z přílohy G byly vybrány doby běhu výsledných kódů vygenerovaných překladači s vybranými úrovněmi optimalizace. Srovnání jednotlivých časů se nachází na obrázku 8.14.

Kód bez optimalizací Neoptimalizovaný kód s nejkratší dobou běhu byl v této úloze vygenerován překladačem *Free Pascal*. Tento kód byl přibližně o 12 % rychlejší než kódy napsané v jazyce *C*. Nejpomalejší kód byl vygenerován překladači *C/C++*, a to po překladu zdrojových kódů napsaných v jazyce *C++*. Důvodem bylo použití kontejneru *std::vector*, který nemohl být optimalizován, čímž docházelo k obrovskému množství funkčních volání.

Kód s bezpečnými optimalizacemi Nejrychlejší kód s bezpečnými optimalizacemi byl vygenerován překladačem *GCC*. Tento překladač zvládnul ze zdrojových souborů napsaných v jazyce C vytvořit kód, který byl téměř o 25 % rychlejší než druhý nejvýkonnější kód vygenerovaný překladačem *ICC*. Zároveň dokázal překladač *GCC* (resp. *G++*) vygenerovat nejrychlejší *C++* kód, který byl ovšem značně pomalejší než jeho ekvivalentní verze napsaná v jazyce *C*. Překladač *ICC* neprovedl *inlining* metod datové struktury



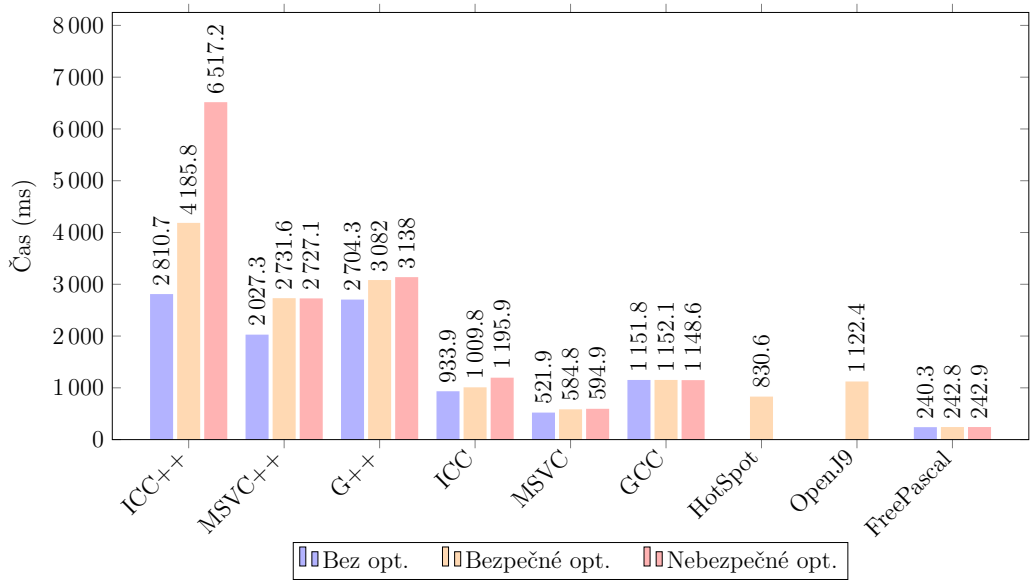
Obrázek 8.14: Srovnání dob běhu překladači vygenerovaných kódů v úloze č. 4, a to v závislosti na zvoleném typu optimalizace.

std::vector, čímž byl výsledný kód, vytvořený ze *C++* zdrojových souborů, značně pomalejší než kódy vygenerované ostatními *C++* překladači. Poměrně rychlý byl také kód vytvořený překladačem *Free Pascal*, který byl v této úloze přibližně dvakrát rychlejší než kódy generované *Java* překladači.

Kód s nebezpečnými optimalizacemi V této úloze neměly nebezpečné optimalizace v podstatě žádný vliv na rychlost doby běhu, a to s výjimkou překladače *ICC*, který provedl s nebezpečnou úrovní *inlining* metod datové struktury *std::vector*, čímž došlo ke značnému zrychlení kódu.

8.4.2 Doba překladu

Statistické veličiny, které byly vypočteny z naměřených časů nutných ke kompilaci zdrojových souborů realizujících úlohu č. 4 se nachází v příloze H. Referenční hodnoty (mediány) byly zaneseny do grafu, který je zobrazen na obrázku 8.15. Na základě těchto hodnot lze konstatovat, že byly nejrychleji přeloženy zdrojové soubory překladačem *Free Pascal*, zatímco nejpomaleji byly přeloženy zdrojové soubory napsané v programovacím jazyce *C++*. Obdobně jako v předchozích úlohách trvala překladači *ICC* kompilace *C++* zdrojových souborů přibližně dvakrát déle (s nebezpečnou úrovní), než ostatním *C++* překladačům.



Obrázek 8.15: Srovnání časů nutných k překladu zdrojových souborů realizujících úlohu č. 4, a to v závislosti na použitém překladači a úrovni optimalizace.

8.4.3 Paměťová náročnost

V tabulce 8.10 se nachází paměťová náročnost jednotlivých výsledných kódů. Na základě dat obsažených v této tabulce lze konstatovat, že stejně jako v předchozích úlohách byl i v této úloze paměťově nejméně náročný kód vygenerován *C/C++* překladači. Výsledný kód generovaný překladačem *Free Pascal* byl ve srovnání s *C/C++* kódy o přibližně 4–5 MB náročnější na paměť. Virtuální stroje *Javy* opět rezervovaly obrovské množství paměti – například virutální stroj *OpenJ9* použil až 530 MB paměti, což je přibližně dvacet pětkrát více, než bylo použito výslednými *C/C++* kódy.

Tabulka 8.10: Paměťová náročnost výsledných kódů realizujících úlohu č. 4, a to v závislosti na použitém překladači a úrovni optimalizace.

Jazyk	Překladač	Použitá paměť (MB)		
		Bez optimalizace	Bezpečná úroveň opt.	Nebezpečná úroveň opt.
C++	ICC	21,85	21,84	21,85
	MSVC	21,83	21,83	21,83
	GCC	21,18	21,18	21,18
C	ICC	21,83	21,83	21,83
	MSVC	21,83	21,82	21,82
	GCC	21,89	21,89	21,89
Java	HotSpot	–	226,66	–
	OpenJ9	–	532,75	–
Object Pascal	FreePascal	26,02	26,02	26,02

9 Zhodnocení výsledků

Na základě provedených testů lze konstatovat, že byly nejrychlejší kódy vygenerovány *C/C++* překladači, které provedly naprosto nejefektivnější optimalizace kódů. Překladač *Free Pascal* sice zvládnul ve třech úlohách vygenerovat mírně rychlejší kód bez optimalizací, jakmile ale došlo k použití optimalizací (bezpečných i nebezpečných), byl kód generovaný tímto překladačem téměř ve všech úlohách naprosto nejpomalejší. S použitím úrovní *O4* a *Ov* sice dokázal překladač *Free Pascal* občas provést některé nebezpečné optimalizace, ty ale neměly téměř žádný vliv na rychlost běhu výsledných kódů, pouze na přesnost výsledku. Zároveň nebyla tímto překladačem provedena vektorizace kódu, a to ani v jedné z úloh. V návaznosti na předchozí nedostatky je nutno zmínit, že je tímto překladačem poskytováno jen velice limitované množství optimalizací, což je pravděpodobně zapříčiněno mnohem menším týmem lidí zabývajících se vývojem daného překladače. Naprosto nejrychlejší kódy s bezpečnými optimalizacemi byly vygenerovány překladačem *MSVC*, a to ve třech ze čtyř úloh. Často byl však tento překladač neefektivní při použití nebezpečných optimalizací, jelikož neprovedl v úlohách č. 1 a 3 vektorizaci kódu, čímž značně zaostával za ostatními *C/C++* překladači. Překladač *ICC* naopak prováděl velice efektivní optimalizace v nebezpečném módu, čímž byly kódy generované tímto překladačem v dané kategorii nejrychlejší – kromě úlohy č. 4, která však neobsahovala operace s čísly v plovoucí řadové čárce.

Překlad zdrojových kódů byl nejrychleji proveden překladačem *Free Pascal*, a to ve všech úlohách. Dokonce i s použitím nejvyšší dostupné úrovně optimalizace došlo jen k velice mírnému nárůstu kompilační doby, která vždy vzrostla jen o pár procent. Na druhou stranu však byly provedené optimalizace ze všech překladačů nejméně efektivní. Velice dobré výsledky vykazoval také překladač *MSVC*, který zvládnul přeložit kód ze všech *C/C++* překladačů nejrychleji a především zdrojové soubory napsané v jazyce *C* byly přeloženy tímto překladačem velice rychle. Nejdéle trval překlad zdrojových souborů napsaných v jazyce *C++*, a to třikrát až sedmkrát déle (v závislosti na překladači a použité úrovni optimalizace) než překlad zdrojových souborů napsaných v *C*.

Paměťově nejméně náročné byly výsledné kódy vytvořené překladači *C/C++*. Kódy vygenerované překladačem *Free Pascal* byly v testovacích úlohách o 0,5 až 5 MB náročnější na paměť, což je stále velice dobré. Naopak nejhorší výsledky byly v této oblasti dosaženy virtuálními stroji *Javy*,

které si zarezervovaly příliš mnoho nepotřebné paměti a byly tak ve srovnání s kódy generovanými ostatními překladači dvacetkrát až čtyřicetkrát náročnější na paměť.

10 Závěr

Zadání bylo splněno v celém rozsahu. Na začátku celé práce, v kapitole 2, byl zadefinován výkon překladače. Následující kapitola 3 se zabývala hardwarovými výkonnostními čítači, které jsou často používány při analýze výkonu aplikací, a to včetně překladačů. V další kapitole byla provedena analýza technik používaných při zkoumání výkonu aplikací, na kterou navazovala kapitola 5, která se věnovala konkrétnímu nástroji určenému k měření výkonu. Následně byly v kapitole 6 navrženy testovací úlohy, které byly použity k objektivnímu porovnání výkonu vybraných překladačů. V kapitole 7 byla popsána metodika měření výsledných kódů, na kterou navazuje kapitola 8, ve které byly jednak porovnány výkony jednotlivých kódů a jednak došlo k hlubší analýze dosažených výsledků. Poslední kapitola obsahuje celkové zhodnocení výkonů překladačů, které se opírá o dosažené výsledky získané během měření.

V návaznosti na tuto práci by mohla být provedena analýza dalších nástrojů určených k měření výkonu aplikací, a to zejména nástrojů používaných na operačním systému *GNU/Linux*. Dále by mohly být navrženy další testovací úlohy zaměřující se na jiné aspekty překladačů, které by mohly být měřeny jednak na jiném zařízení a jednak také na jiném operačním systému.

Literatura

[1] *AMD uProf User Guide* [online]. Advanced Micro Devices, Inc., 2019. [cit. 11.11.2019]. Dostupné z: https://developer.amd.com/wordpress/media/2013/12/User_Guide.pdf.

[2] *Fundamentals of Performance Profiling* [online]. [cit. 8.1.2020]. Dostupné z: https://smartbear.com/learn/code-profiling/fundamentals-of-performance-profiling/.

[3] *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B* [online]. Intel, 2016. [cit. 20.2.2020]. Dostupné z: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.pdf.

[4] *Intel® Microarchitecture Codename Nehalem Performance Monitoring Unit Programming Guide* [online]. Intel, 2010. [cit. 15.12.2019]. Dostupné z: https://software.intel.com/sites/default/files/76/87/30320.

[5] *Timestamping* [online]. [cit. 3.2.2020]. Dostupné z: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_for_real_time/7/html/reference_guide/chap-timestamping.

[6] Aman Singh, A. B. *A Study of Performance Monitoring Unit, perf and perf_events subsystem* [online]. [cit. 19.12.2019]. Dostupné z: http://rts.lab.asu.edu/web_438/project_final/CSE_598_Performance_Monitoring_Unit.pdf.

[7] Bakhvalov, D. *Advanced profiling topics. PEBS and LBR.* [online]. 2018. [cit. 8.12.2019]. Dostupné z: https://easyperf.net/blog/2018/06/08/Advanced-profiling-topics-PEBS-and-LBR#last-branch-record-lbr.

[8] Drongowski, P. J. *Instruction-Based Sampling: A New Performance Analysis Technique for AMD Family 10h Processors* [online]. Advanced Micro Devices, Inc., 2007. [cit. 11.11.2019]. Dostupné z: https://developer.amd.com/wordpress/media/2012/10/AMD_IBS_paper_EN.pdf.

[9] Graham, S. *Tim* [online]. GitHub, 2017. [cit. 21.1.2020]. Dostupné z: https://github.com/sgraham/tim.

[10] *Time Measurement* [online]. Yandex LLC. [cit. 19.12.2019]. Dostupné z: https://handystats.readthedocs.io/en/latest/time-measurement.html.

[11] Kukunas, J. *Power and Performance (1st Ed.): Software Analysis and Optimization*. Morgan Kaufmann Publishers, 2015. ISBN 0-128-00726-5.

[12] Mallony, A. D. – Mellor-Crummey, J. – Shende, S. S. Measurement and Analysis of Parallel Program Performance Using TAU and HPCToolkit. In David, B. – Robert, L. – Samuel, W. (Ed.) *Peformance Tuning of Scientific Applications*. Boca Raton, Florida, USA: Chapman & Hall/CRC Computational Science, 2015. Kapitola 4, s. 49–86.

[13] Marusarz, J. – Dmitry, R. *Top-down Microarchitecture Analysis Method* [online]. Intel, 2020. [cit. 21.1.2020]. Intel VTune Profiler Performance Analysis Cookbook. Dostupné z: `https://software.intel.com/content/www/us/en/develop/documentation/vtune-cookbook/top/methodologies/top-down-microarchitecture-analysis-method.html`.

[14] Moore, S. V. – Terpstra, D. K. – Weaver, V. M. Software Interfaces to Hardware Counters. In David, B. – Robert, L. – Samuel, W. (Ed.) *Peformance Tuning of Scientific Applications (1st Ed.)*. Boca Raton, Florida, USA: Chapman & Hall/CRC Computational Science, 2010. Kapitola 3, s. 33–48.

[15] Paoloni, G. *How to Benchmark Code Execution Times on Intel® IA-32 and IA-64 Instruction Set Architectures* [online]. Intel Corporation, 2010. [cit. 6.1.2020]. Dostupné z: `https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf`.

[16] Parker, S. et al. Performance Analysis and Debugging Tools at Scale. In Straatsma, T. P. – Antypas, K. B. – Williams, T. J. (Ed.) *Exascale Scientific Applications (1st Ed.): Scalability and Performance Portability*. Boca Raton, Florida, USA: Chapman & Hall/CRC Computational Science, 2017. Kapitola 2, s. 17–49.

[17] Sprunt, B. The basics of performance-monitoring hardware. *IEEE Micro*. 2002, 22, 4, s. 64–71.

[18] Steijn, P. *How we test the compiler performance* [online]. Microsoft, 2010. [cit. 15.1.2020]. Dostupné z: `https://devblogs.microsoft.com/cppblog/how-we-test-the-compiler-performance/`.

[19] *Acquiring high-resolution time stamps* [online]. Microsoft, 2018. [cit. 15.1.2020]. Dostupné z: `https://docs.microsoft.com/en-us/windows/win32/sysinfo/acquiring-high-resolution-time-stamps`.

[20] Yang, O. *Pitfalls of TSC usage* [online]. 2015. [cit. 11.12.2019]. Dostupné z: `http://oliveryang.net/2015/09/pitfalls-of-TSC-usage/`.

Seznam zkratek

PMU	Performance Monitoring Unit – jednotka v procesoru, která slouží k měření výkonu
IRQ	Interrupt Request – žádost o hardwarové přerušení
TSC	Time Stamp Counter – registr v procesoru, který složí k měření hodinových cyklů
RDTSC	Read Time–Stamp Counter – instrukce pro přečtení hodnoty registru TSC
RDTSCP	Read Time-Stamp Counter and Processor ID - „pseudo“ serializační instrukce, slouží k přečtení hodnoty registru TSC a ID procesoru
QPC	QueryPerformanceCounter – funkce pro měření času na operačním systému Windows
QPF	QueryPerformanceFrequency – funkce pro získání frekvence časovače použitého pro měření času pomocí QPC
EBS	Event Based Sampling – vzorkování využívající vlastnosti přetečení hardwarových výkonnostních čítačů
PEBS	Precise Event Based Sampling – přesnější forma EBS
IBS	Instruction Based Sampling – instrukční vzorkování
MSVC	Microsoft Visual C++ – C/C++ překladač od společnosti Microsoft
ICC	Intel C++ Compiler – C/C++ překladač od společnosti Intel
GCC	GNU Compiler Collection – sada překladačů vytvořených v rámci projektu GNU
uOp	Micro-operation – Nejzákladnější operace proveditelná procesorem, instrukce je typicky množinou několika mikrooperací.

Seznam obrázků

Seznam tabulek

Seznam výpisů

Přílohy

A Úloha č. 1 – doby běhu

Tabulka A.1: Statistické veličiny vypočtené z časů potřebných k vykonání výsledných kódů napsaných v jazyce C++, přeložených překladačem ICC a realizujících úlohu č. 1.

Optimalizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
Od	50	1171,8	1496,1	1189,7	1173,1	1175,8	1180,3	59,6	5,00
Os	50	404,4	455,0	408,0	405,9	406,5	407,6	7,0	1,73
O1	50	401,6	453,0	413,7	403,0	403,9	408,7	18,4	4,45
O2	50	232,9	269,2	236,8	233,7	234,4	235,5	6,9	2,91
O3	50	231,5	268,3	234,8	232,2	232,6	234,7	6,3	2,68
Ov	50	227,8	232,5	228,9	228,3	228,9	229,2	7,8	0,34

Tabulka A.2: Statistické veličiny vypočtené z časů potřebných k vykonání výsledných kódů napsaných v jazyce C++, přeložených překladačem MSVC a realizujících úlohu č. 1.

Optimalizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
Od	50	1179,3	1510,6	1198,9	1183,1	1185,2	1189,4	62,7	5,23
O1	50	393,0	489,5	409,3	394,7	397,6	419,9	20,8	5,09
O2	50	393,4	489,3	400,0	394,4	395,0	396,0	18,6	4,66
Ov	50	394,8	489,6	400,9	395,6	396,0	396,8	18,4	4,60

Tabulka A.3: Statistické veličiny vypočtené z časů potřebných k vykonání výsledných kódů napsaných v jazyce C++, přeložených překladačem G++ a realizujících úlohu č. 1.

Optima-lizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
O0	50	1153,8	1453,5	1174,7	1155,9	1159,3	1162,8	54,6	4,65
Os	50	407,0	416,6	409,2	408,1	408,7	409,4	1,7	0,42
O1	50	404,5	497,0	411,8	406,4	407,3	408,4	17,6	4,28
O2	50	405,7	417,0	408,0	407,0	407,5	408,3	2,0	0,48
O3	50	404,8	413,9	407,2	406,1	406,7	407,5	2,0	0,48
Ofast	50	323,8	342,3	326,1	324,6	325,3	326,5	3,0	0,92
Ov	50	237,8	310,0	245,1	239,0	240,0	240,3	17,6	7,20

Tabulka A.4: Statistické veličiny vypočtené z časů potřebných k vykonání výsledných kódů napsaných v jazyce C, přeložených překladačem ICC a realizujících úlohu č. 1.

Optima-lizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
Od	50	1171,8	1496,1	1189,7	1173,1	1175,8	1180,3	59,6	5,00
Os	50	404,4	455,0	408,0	405,9	406,5	407,6	7,0	1,73
O1	50	401,6	453,0	413,7	403,0	403,9	408,7	18,4	4,45
O2	50	232,9	269,2	236,8	233,7	234,4	235,5	6,9	2,91
O3	50	231,5	268,3	234,8	232,2	232,6	234,7	6,3	2,68
Ov	50	227,8	232,5	228,9	228,3	228,9	229,2	7,8	0,34

Tabulka A.5: Statistické veličiny vypočtené z časů potřebných k vykonání výsledných kódů napsaných v jazyce C, přeložených překladačem MSVC a realizujících úlohu č. 1.

Optima-lizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
Od	50	1179,3	1510,6	1198,9	1183,1	1185,2	1189,4	62,7	5,23
O1	50	393,0	489,5	409,3	394,7	397,6	419,9	20,8	5,09
O2	50	393,4	489,3	400,0	394,4	395,0	396,0	18,6	4,66
Ov	50	394,8	489,6	400,9	395,6	396,0	396,8	18,4	4,60

Tabulka A.6: Statistické veličiny vypočtené z časů potřebných k vykonání výsledných kódů napsaných v jazyce C, přeložených překladačem GCC a realizujících úlohu č. 1.

Optimalizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
O0	50	1153,8	1453,5	1174,7	1155,9	1159,3	1162,8	54,6	4,65
Os	50	407,0	416,6	409,2	408,1	408,7	409,4	1,7	0,42
O1	50	404,5	497,0	411,8	406,4	407,3	408,4	17,6	4,28
O2	50	405,7	417,0	408,0	407,0	407,5	408,3	2,0	0,48
O3	50	404,8	413,9	407,2	406,1	406,7	407,5	2,0	0,48
Ofast	50	323,8	342,3	326,1	324,6	325,3	326,5	3,0	0,92
Ov	50	237,8	310,0	245,1	239,0	240,0	240,3	17,6	7,20

Tabulka A.7: Statistické veličiny vypočtené z časů potřebných k vykonání výsledných kódů napsaných v jazyce Java, vykonaných virtuálními stroji Javy a realizujících úlohu č. 1.

JVM	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
HotSpot	50	448,2	673,9	460,7	451,4	452,4	456,1	35,4	7,68
OpenJ9	50	436,2	1376,7	479,8	437,8	438,7	439,6	163,1	33,90

Tabulka A.8: Statistické veličiny vypočtené z časů potřebných k vykonání výsledných kódů napsaných v jazyce Object Pascal, přeložených překladačem FreePascal a realizujících úlohu č. 1.

Optimalizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
O0	50	1051,5	1377,8	1069,6	1053,3	1054,6	1057,9	59,4	5,55
Os	50	1059,8	1392,8	1080,4	1060,7	1061,9	1066,3	61,9	5,72
O1	50	1063,9	1367,0	1083,0	1066,0	1068,5	1072,6	52,4	4,84
O2	50	530,7	767,8	542,8	532,0	532,7	537,6	42,3	7,78
O3	50	538,7	839,8	550,8	540,0	541,0	545,8	44,6	8,10
O4	50	525,4	848,1	536,6	526,6	527,8	532,1	45,6	8,49
Ov	50	551,4	841,6	562,8	552,4	553,5	557,8	41,6	7,39

Tabulka A.9: Statistické veličiny vypočtené z časů potřebných k vykonání výsledného kódu napsaného v jazyce C++, přeloženého překladačem ICC a realizujícího úlohu č. 1 s použitím restrict ukazatelů.

Optimalizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
Ov	50	43,3	47,3	43,6	43,4	43,5	43,6	0,6	1,31

Tabulka A.10: Statistické veličiny vypočtené z časů potřebných k vykonání výsledného kódu napsaného v jazyce C, přeloženého překladačem ICC a realizujícího úlohu č. 1 s použitím restrict ukazatelů.

Optimalizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
Ov	50	42,9	44,5	43,2	43,0	43,1	43,3	0,3	0,76

B Úloha č. 1 – doby překladu

Tabulka B.1: Statistické veličiny vypočtené z časů nutných k překladu kódů napsaných v jazyce C++, přeložených překladačem ICC a realizujících úlohu č. 1.

Optimalizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
Od	20	3339,6	3590,9	3369,9	3349,2	3351,3	3362,2	54,5	1,62
O1	20	4794,2	5611,2	4857,7	4803,4	4809,2	4838,2	178,6	3,68
Ov	20	7783,0	8149,7	7840,1	7793,1	7813,3	7837,2	89,0	1,13

Tabulka B.2: Statistické veličiny vypočtené z časů nutných k překladu kódů napsaných v jazyce C++, přeložených překladačem MSVC a realizujících úlohu č. 1.

Optimalizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
Od	20	2436,5	2683,6	2462,5	2441,3	2446,2	2454,1	54,4	2,21
O2	20	3248,3	3532,9	3331,2	3272,5	3282,0	3395,9	87,6	2,63
Ov	20	3263,0	3828,1	3329,1	3284,7	3296,1	3320,9	120,5	3,62

Tabulka B.3: Statistické veličiny vypočtené z časů nutných k překladu kódů napsaných v jazyce C++, přeložených překladačem G++ a realizujících úlohu č. 1.

Optimalizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
O0	20	3158,1	3244,7	3170,3	3160,4	3162,5	3168,4	21,2	0,67
O2	20	3656,7	3722,3	3669,2	3659,1	3663,9	3667,9	18,3	0,50
Ov	20	3714,3	3784,2	3730,9	3719,7	3723,3	3731,7	19,8	0,53

Tabulka B.4: Statistické veličiny vypočtené z časů nutných k překladu kódů napsaných v jazyce C, přeložených překladačem ICC a realizujících úlohu č. 1.

Optimalizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
Od	30	947,2	1389,5	1007,3	955,7	963,0	981,5	111,0	11,02
O1	30	1040,6	1071,6	1047,0	1042,6	1045,6	1048,7	6,5	0,62
Ov	30	1219,4	1435,3	1238,2	1224,3	1226,5	1232,6	39,9	3,22

Tabulka B.5: Statistické veličiny vypočtené z časů nutných k překladu kódů napsaných v jazyce C, přeložených překladačem MSVC a realizujících úlohu č. 1.

Optimalizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
Od	30	535,1	863,4	558,1	541,3	543,5	551,9	58,4	10,46
O2	30	617,4	973,3	636,5	621,3	624,6	627,2	63,8	10,03
Ov	30	616,4	665,9	622,2	619,3	620,9	621,7	8,7	1,40

Tabulka B.6: Statistické veličiny vypočtené z časů nutných k překladu kódů napsaných v jazyce C, přeložených překladačem GCC a realizujících úlohu č. 1.

Optimalizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
O0	30	1036,7	1089,4	1042,3	1038,7	1039,3	1040,7	10,1	0,97
O3	30	1229,0	1521,7	1256,1	1230,4	1232,5	1269,9	57,3	4,56
Ov	30	1234,0	1290,4	1239,4	1236,0	1237,0	1238,2	10,4	0,83

Tabulka B.7: Statistické veličiny vypočtené z časů nutných k překladu kódů napsaných v jazyce Java, přeložených Java překladači a realizujících úlohu č. 1.

JVM	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
HotSpot	30	842,7	892,3	864,8	859,7	861,5	874,2	11,5	1,33
OpenJ9	30	1141,0	1559,8	1172,7	1153,9	1157,6	1162,2	74,1	6,32

Tabulka B.8: Statistické veličiny vypočtené z časů nutných k překladu kódů napsaných v jazyce Object Pascal, přeložených překladačem FreePascal a realizujících úlohu č. 1.

Optima-lizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
O0	30	246,7	290,8	257,1	250,2	251,4	255,6	13,2	5,14
O2	30	252,3	299,4	257,8	253,7	255,0	257,7	9,9	3,85
O4	30	250,5	329,8	264,3	254,9	256,6	260,8	21,6	8,19

C Úloha č. 2 – doby běhu

Tabulka C.1: Statistické veličiny vypočtené z časů potřebných k vykonání výsledných kódů napsaných v jazyce C++, přeložených překladačem ICC a realizujících úlohu č. 2.

Optimalizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
Od	50	903,7	916,1	905,8	904,0	904,2	908,3	2,7	0,30
Os	50	188,7	192,8	189,2	188,8	189,0	189,3	0,6	0,33
O1	50	215,4	230,2	216,8	215,9	216,2	216,7	2,4	1,09
O2	50	92,8	97,5	93,6	93,0	93,2	93,8	0,9	0,92
O3	50	93,2	97,5	94,0	93,4	93,8	94,4	0,8	0,87
Ov	50	61,3	66,3	62,1	61,6	61,7	62,1	0,9	1,49

Tabulka C.2: Statistické veličiny vypočtené z časů potřebných k vykonání výsledných kódů napsaných v jazyce C++, přeložených překladačem MSVC a realizujících úlohu č. 2.

Optimalizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
Od	50	887,5	918,6	890,9	887,9	888,2	892,0	6,5	0,73
O1	50	213,9	220,9	214,7	214,1	214,4	214,7	1,4	0,66
O2	50	190,8	191,7	191,1	191,0	191,1	191,2	0,2	0,09
Ov	50	69,3	75,7	70,6	69,9	70,4	70,9	1,1	1,55

Tabulka C.3: Statistické veličiny vypočtené z časů potřebných k vykonání výsledných kódů napsaných v jazyce C++, přeložených překladačem G++ a realizujících úlohu č. 2.

Optima-lizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
O0	50	906,9	957,0	911,9	907,2	907,4	908,9	10,6	1,16
Os	50	216,8	217,8	217,1	217,0	217,2	217,3	0,2	0,10
O1	50	220,4	227,4	221,7	220,6	220,8	221,0	2,2	0,98
O2	50	208,5	240,4	209,8	208,6	208,8	209,0	4,7	2,24
O3	50	207,4	228,1	208,5	207,6	207,8	208,1	3,1	1,48
Ofast	50	196,4	200,0	197,2	196,7	197,0	197,3	0,7	0,37
Ov	50	94,5	97,2	95,1	94,7	94,9	95,2	0,6	0,61

Tabulka C.4: Statistické veličiny vypočtené z časů potřebných k vykonání výsledných kódů napsaných v jazyce C, přeložených překladačem ICC a realizujících úlohu č. 2.

Optima-lizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
Od	50	904,3	930,5	906,8	904,6	904,7	905,6	5,0	0,55
Os	50	189,4	196,9	190,2	189,6	190,0	190,3	1,2	0,62
O1	50	191,9	209,4	202,8	200,9	203,1	204,6	3,5	1,72
O2	50	93,5	99,4	94,5	93,8	94,0	94,9	1,2	1,31
O3	50	93,1	96,6	93,8	93,3	93,5	94,1	0,7	0,70
Ov	50	61,3	65,1	62,0	61,5	61,7	62,0	0,8	1,27

Tabulka C.5: Statistické veličiny vypočtené z časů potřebných k vykonání výsledných kódů napsaných v jazyce C, přeložených překladačem MSVC a realizujících úlohu č. 2.

Optima-lizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
Od	50	887,9	921,7	890,3	888,1	888,3	889,9	5,6	0,63
O1	50	216,9	229,8	217,6	217,0	217,3	217,5	1,8	0,84
O2	50	185,6	191,6	186,3	185,8	185,9	186,1	1,5	0,79
Ov	50	70,1	73,3	71,1	70,6	71,0	71,5	0,7	0,98

Tabulka C.6: Statistické veličiny vypočtené z časů potřebných k vykonání výsledných kódů napsaných v jazyce C, přeložených překladačem GCC a realizujících úlohu č. 2.

Optimalizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
O0	50	908,7	920,9	910,4	909,0	909,1	910,3	2,5	0,27
Os	50	212,0	220,6	213,1	212,2	212,5	212,6	1,9	0,91
O1	50	201,3	211,4	206,6	205,3	206,6	207,6	2,1	1,03
O2	50	193,6	211,5	194,9	193,8	194,0	194,1	3,1	1,57
O3	50	207,0	208,4	207,2	207,1	207,1	207,2	0,2	0,11
Ofast	50	196,1	198,5	196,6	196,3	196,5	196,8	0,4	0,22
Ov	50	92,2	94,3	92,6	92,3	92,4	92,9	0,5	0,51

Tabulka C.7: Statistické veličiny vypočtené z časů potřebných k vykonání výsledných kódů napsaných v jazyce Java, přeložených Java překladači a realizujících úlohu č. 2.

JVM	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
HotSpot	50	282,9	347,4	286,3	283,1	283,4	283,7	12,6	4,39
OpenJ9	50	353,6	1225,7	388,8	353,8	354,0	354,4	145,2	37,34

Tabulka C.8: Statistické veličiny vypočtené z časů potřebných k vykonání výsledných kódů napsaných v jazyce Object Pascal, přeložených překladačem FreePascal a realizujících úlohu č. 2.

Optimalizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
O0	50	924,8	938,9	927,7	925,1	925,5	930,6	3,3	0,36
Os	50	928,0	948,3	931,3	928,3	928,5	933,5	5,0	0,53
O1	50	924,4	950,9	926,9	924,7	925,0	925,9	4,5	0,48
O2	50	428,9	440,9	432,0	429,9	430,6	431,2	3,7	0,85
O3	50	391,8	399,7	393,4	392,2	392,3	392,7	2,4	0,61
O4	50	392,1	404,8	393,8	392,6	392,8	393,5	2,7	0,69
Ov	50	392,4	413,7	394,7	393,0	393,2	393,6	4,5	1,15

Tabulka C.9: Statistické veličiny vypočtené z časů potřebných k vykonání výsledných kódů napsaných v jazyce C++, přeložených překladačem ICC a realizujících úlohu č. 2 s použitím gen. kontejneru.

Optimalizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
Od	10	7238,8	7256,0	7247,6	7243,4	7247,4	7250,9	5,3	0,07
O1	50	208,0	214,4	208,4	208,2	208,3	208,4	0,9	0,42
Ov	50	62,1	67,6	63,0	62,4	62,7	63,1	1,0	1,51

Tabulka C.10: Statistické veličiny vypočtené z časů potřebných k vykonání výsledných kódů napsaných v jazyce C++, přeložených překladačem MSVC a realizujících úlohu č. 2 s použitím gen. kontejneru.

Optimalizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
Od	10	4767,3	4784,7	4771,1	4768,2	4769,5	4772,3	5,1	0,11
O2	50	231,6	239,3	232,4	231,8	232,0	232,1	1,6	0,69
Ov	50	71,4	77,1	72,5	71,8	72,2	72,8	1,2	1,59

Tabulka C.11: Statistické veličiny vypočtené z časů potřebných k vykonání výsledných kódů napsaných v jazyce C++, přeložených překladačem G++ a realizujících úlohu č. 2 s použitím gen. kontejneru.

Optimalizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
O0	10	1738,2	1804,1	1761,1	1744,8	1759,3	1777,0	21,0	1,19
O3	50	235,8	253,1	237,5	236,0	236,2	236,4	3,4	1,41
Ov	50	96,2	98,1	96,7	96,4	96,6	97,1	0,5	0,51

Tabulka C.12: Statistické veličiny vypočtené z časů potřebných k vykonání výsledných kódů napsaných v jazyce Java, přeložených Java překladači a realizujících úlohu č. 2 s použitím gen. kontejneru.

JVM	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
HotSpot	50	731,1	832,0	751,3	735,2	736,7	746,7	28,3	3,76
OpenJ9	50	2508,1	4393,3	2619,6	2510,2	2518,4	2533,9	352,0	13,44

Tabulka C.13: Statistické veličiny vypočtené z časů potřebných k vykonání výsledných kódů napsaných v jazyce Object Pascal, přeložených překladačem FreePascal a realizujících úlohu č. 2 s použitím gen. kontejneru.

Optimalizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
O0	50	2030,0	2058,9	2037,2	2031,9	2036,3	2041,4	5,9	0,29
O3	50	963,8	971,7	965,6	964,2	964,4	965,3	2,4	0,25
O4	50	1036,5	1049,2	1038,6	1036,7	1037,4	1040,0	2,6	0,25

D Úloha č. 2 – doby překladu

Tabulka D.1: Statistické veličiny vypočtené z časů nutných k překladu kódů napsaných v jazyce C++, přeložených překladačem ICC a realizujících úlohu č. 2

Optimalizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
Od	20	3346,7	3864,7	3415,0	3354,2	3359,9	3401,3	129,7	3,80
O1	20	4854,8	5358,0	4936,3	4865,9	4877,8	4936,6	126,5	2,56
Ov	20	7879,0	8250,2	7994,2	7907,2	7943,8	8051,6	122,0	1,53

Tabulka D.2: Statistické veličiny vypočtené z časů nutných k překladu kódů napsaných v jazyce C++, přeložených překladačem MSVC a realizujících úlohu č. 2.

Optimalizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
Od	20	2455,6	2612,0	2510,3	2477,8	2506,1	2531,6	41,3	1,64
O2	20	3300,5	3546,6	3364,9	3316,7	3351,0	3375,3	64,6	1,92
Ov	20	3321,7	3766,1	3392,8	3334,6	3375,9	3406,3	97,5	2,87

Tabulka D.3: Statistické veličiny vypočtené z časů nutných k překladu kódů napsaných v jazyce C++, přeložených překladačem G++ a realizujících úlohu č. 2.

Optimalizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
O0	20	3171,5	3300,0	3191,3	3174,4	3179,3	3186,4	34,6	1,08
O3	20	3800,6	3884,3	3814,4	3803,9	3808,7	3816,9	18,7	0,49
Ov	20	3803,0	4139,8	3834,7	3804,7	3809,2	3820,8	75,3	1,96

Tabulka D.4: Statistické veličiny vypočtené z časů nutných k překladu kódů napsaných v jazyce C, přeložených překladačem ICC a realizujících úlohu č. 2.

Optimalizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
Od	30	965,7	1148,6	980,9	969,1	971,7	977,0	34,9	3,56
O1	30	1067,1	1353,4	1090,4	1068,6	1072,2	1076,4	67,9	6,23
Ov	30	1256,1	1473,7	1280,8	1266,9	1270,5	1281,3	38,2	2,98

Tabulka D.5: Statistické veličiny vypočtené z časů nutných k překladu kódů napsaných v jazyce C, přeložených překladačem MSVC a realizujících úlohu č. 2.

Optimalizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
Od	30	551,1	799,4	591,6	555,4	565,4	600,0	56,4	9,52
O2	30	636,6	957,8	691,7	649,7	675,3	704,2	68,6	9,91
Ov	30	649,7	983,2	707,4	664,5	691,2	717,2	70,5	9,97

Tabulka D.6: Statistické veličiny vypočtené z časů nutných k překladu kódů napsaných v jazyce C, přeložených překladačem GCC a realizujících úlohu č. 2.

Optimalizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
O0	30	1047,7	1160,4	1055,5	1049,3	1049,9	1051,4	21,2	2,01
O2	30	1225,6	1249,0	1231,2	1228,5	1230,1	1232,1	4,6	0,37
Ov	30	1272,0	1476,7	1282,8	1273,6	1275,0	1276,2	37,1	2,89

Tabulka D.7: Statistické veličiny vypočtené z časů nutných k překladu kódů napsaných v jazyce Java, přeložených Java překladači a realizujících úlohu č. 2.

JVM	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
HotSpot	30	842,7	892,3	864,8	859,7	861,5	874,2	11,5	1,33
OpenJ9	30	1141,0	1559,8	1172,7	1153,9	1157,6	1162,2	74,1	6,32

Tabulka D.8: Statistické veličiny vypočtené z časů nutných k překladu kódů napsaných v jazyce Object Pascal, přeložených překladačem FreePascal a realizujících úlohu č. 2.

Optima-lizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
O0	20	245,5	344,4	256,8	246,6	247,8	249,6	23,0	8,95
O3	20	251,5	558,7	266,8	252,9	254,1	256,7	55,7	20,87
O4	20	251,2	309,4	260,2	252,1	253,2	261,2	14,6	5,62

E Úloha č. 3 – doby běhu

Tabulka E.1: Statistické veličiny vypočtené z časů potřebných k vykonání výsledných kódů napsaných v jazyce C++, přeložených překladačem ICC a realizujících úlohu č. 3.

Optimalizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
Od	50	1361,4	1383,6	1363,6	1362,0	1362,7	1363,4	4,0	0,29
Os	50	264,0	285,8	267,5	264,4	265,2	267,0	5,8	2,15
O1	50	288,0	292,5	289,6	288,6	289,6	290,2	1,1	0,37
O2	50	188,8	198,3	191,6	189,7	191,0	193,0	2,2	1,14
O3	50	187,2	206,5	190,4	187,8	189,2	190,6	4,2	2,19
Ov	50	142,3	153,3	144,2	142,8	143,5	145,7	2,0	1,39

Tabulka E.2: Statistické veličiny vypočtené z časů potřebných k vykonání výsledných kódů napsaných v jazyce C++, přeložených překladačem MSVC a realizujících úlohu č. 3.

Optimalizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
Od	50	1376,2	1380,5	1377,1	1376,5	1376,8	1377,3	0,9	0,06
O1	50	291,3	311,8	293,8	292,2	292,7	293,4	3,8	1,28
O2	50	247,1	260,5	249,7	248,4	249,1	250,1	2,3	0,93
Ov	50	261,8	272,6	263,3	262,3	262,8	263,6	1,7	0,66

Tabulka E.3: Statistické veličiny vypočtené z časů potřebných k vykonání výsledných kódů napsaných v jazyce C++, přeložených překladačem G++ a realizujících úlohu č. 3.

Optimalizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
O0	50	1300,0	1308,3	1302,1	1301,1	1301,9	1302,5	1,5	0,12
Os	50	290,4	296,7	292,0	291,1	291,6	292,5	1,4	0,47
O1	50	264,6	287,3	266,8	265,6	266,3	266,9	3,2	1,18
O2	50	287,5	295,9	289,1	288,4	288,8	289,6	1,4	0,49
O3	50	255,0	276,4	257,5	255,9	256,7	257,5	3,5	1,35
Ofast	50	252,0	262,9	254,2	252,9	253,6	254,7	2,0	0,79
Ov	50	152,9	174,3	155,2	153,3	153,9	156,2	3,5	2,28

Tabulka E.4: Statistické veličiny vypočtené z časů potřebných k vykonání výsledných kódů napsaných v jazyce C, přeložených překladačem ICC a realizujících úlohu č. 3.

Optimalizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
Od	50	1361,1	1365,4	1362,5	1361,8	1362,2	1363	1,0	0,07
Os	50	286,8	319,7	289,5	287,5	288,2	289,4	5,5	1,89
O1	50	287,3	294,4	288,7	287,9	288,5	289,3	1,2	0,43
O2	50	188,4	197,3	190,4	189,1	190,0	191,2	1,8	0,92
O3	50	188,8	198,6	191,0	189,6	190,4	192,1	1,9	0,98
Ov	50	142,6	156,9	145,0	143,2	144,4	146,2	2,4	1,68

Tabulka E.5: Statistické veličiny vypočtené z časů potřebných k vykonání výsledných kódů napsaných v jazyce C, přeložených překladačem MSVC a realizujících úlohu č. 3.

Optimalizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
Od	50	1373,1	1378,0	1374,7	1374,0	1374,4	1375,5	1,1	0,08
O1	50	285,4	291,7	287,0	286,2	286,6	287,4	1,3	0,44
O2	50	245,5	258,3	248,0	246,6	247,7	248,6	2,1	0,83
Ov	50	260,0	265,6	262,0	261,1	262,0	262,6	1,2	0,45

Tabulka E.6: Statistické veličiny vypočtené z časů potřebných k vykonání výsledných kódů napsaných v jazyce C, přeložených překladačem GCC a realizujících úlohu č. 3.

Optimalizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
O0	50	1318,4	1358,6	1320,9	1319,1	1319,4	1321,2	5,7	0,43
Os	50	289,4	298,8	291,7	290,2	291,1	292,1	2,3	0,79
O1	50	266,3	290,0	270,9	267,6	268,8	269,7	6,4	2,35
O2	50	286,3	291,5	287,8	286,9	287,7	288,5	1,1	0,39
O3	50	258,0	269,0	260,1	258,8	259,8	260,8	2,1	0,80
Ofast	50	257,3	264,6	259,2	257,9	258,7	259,8	1,7	0,65
Ov	50	154,4	164,5	156,1	154,7	155,6	156,8	1,9	1,24

Tabulka E.7: Statistické veličiny vypočtené z časů potřebných k vykonání výsledných kódů napsaných v jazyce Java, přeložených Java překladači a realizujících úlohu č. 3.

JVM	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
HotSpot	50	327,2	383,8	331,5	328,4	329,1	330,6	9,2	2,78
OpenJ9	50	256,6	1637,9	313,2	257,8	258,5	259,5	225,5	71,99

Tabulka E.8: Statistické veličiny vypočtené z časů potřebných k vykonání výsledných kódů napsaných v jazyce Object Pascal, přeložených překladačem FreePascal a realizujících úlohu č. 3.

Optimalizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
O0	50	1286,7	1357,5	1292,7	1287,2	1287,6	1288,7	14,7	1,14
Os	50	1287,9	1345,7	1292,6	1288,2	1288,4	1289,1	13,4	1,04
O1	50	1301,0	1358,3	1305,7	1301,3	1301,4	1302,1	12,4	0,95
O2	50	740,1	779,9	742,9	740,6	741,7	742,3	5,7	0,76
O3	50	707,4	747,2	712,4	707,9	708,3	709,1	10,0	1,40
O4	50	707,2	755,5	713,7	707,5	708,0	709,1	13,6	1,91
Ov	50	716,7	728,3	718,1	717,2	717,4	718,1	2,0	0,27

Tabulka E.9: Statistické veličiny vypočtené z časů potřebných k vykonání výsledných kódů napsaných v jazyce C++, přeložených překladačem ICC a realizujících úlohu č. 3 s použitím datového typu float.

Optimalizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
Od	50	1344,7	1412,6	1350,7	1345,0	1345,2	1347,0	15,2	1,13
O1	50	237,6	243,6	239,6	239,0	239,2	239,6	1,1	0,47
Ov	50	53,7	57,8	54,0	53,8	53,8	53,9	0,6	1,08

Tabulka E.10: Statistické veličiny vypočtené z časů potřebných k vykonání výsledných kódů napsaných v jazyce C++, přeložených překladačem MSVC a realizujících úlohu č. 3 s použitím datového typu float.

Optimalizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
Od	50	1357,9	1430,8	1364,7	1358,2	1359,4	1362,1	15,5	1,13
O2	50	192,9	194,7	193,3	193,1	193,2	193,4	0,4	0,19
Ov	50	199,4	212,6	200,4	199,8	199,9	200,1	2,1	1,02

Tabulka E.11: Statistické veličiny vypočtené z časů potřebných k vykonání výsledných kódů napsaných v jazyce C++, přeložených překladačem G++ a realizujících úlohu č. 3 s použitím datového typu float.

Optimalizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
O0	50	1280,3	1372,9	1288,2	1280,6	1280,9	1284,2	19,5	1,51
O3	50	139,9	144,7	140,9	140,5	140,6	141,0	0,9	0,60
Ov	50	62,6	64,9	63,1	62,7	63,1	63,3	0,4	0,66

Tabulka E.12: Statistické veličiny vypočtené z časů potřebných k vykonání výsledných kódů napsaných v jazyce C, přeložených překladačem ICC a realizujících úlohu č. 3 s použitím datového typu float.

Optimalizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
Od	50	1343,7	1407,2	1349,6	1344,0	1344,2	1345,1	14,0	1,04
O1	50	237,5	270,9	239,0	237,8	238,0	238,7	4,7	1,95
Ov	50	54,4	56,5	54,6	54,5	54,6	54,7	0,3	0,57

Tabulka E.13: Statistické veličiny vypočtené z časů potřebných k vykonání výsledných kódů napsaných v jazyce C, přeložených překladačem MSVC a realizujících úlohu č. 3 s použitím datového typu float.

Optimalizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
Od	50	1355,8	1396,9	1359,7	1356,0	1356,1	1356,3	10,7	0,79
O2	50	193,5	228,2	195,5	193,6	193,7	194,2	5,5	2,79
Ov	50	199,9	237,1	201,8	200,2	200,5	200,7	6,1	3,00

Tabulka E.14: Statistické veličiny vypočtené z časů potřebných k vykonání výsledných kódů napsaných v jazyce C, přeložených překladačem GCC a realizujících úlohu č. 3 s použitím datového typu float.

Optimalizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
O0	50	1301,6	1354,4	1305,9	1302,0	1302,6	1303,1	11,4	0,87
O3	50	139,6	147,7	140,4	140,1	140,2	140,5	1,1	0,77
Ov	50	62,6	63,6	63,1	62,8	63,1	63,2	0,3	0,44

Tabulka E.15: Statistické veličiny vypočtené z časů potřebných k vykonání výsledných kódů napsaných v jazyce Java, přeložených Java překladači a realizujících úlohu č. 3 s použitím datového typu float.

JVM	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
HotSpot	50	291,0	319,5	293,1	291,5	291,8	292,7	4,3	1,45
OpenJ9	50	205,2	1433,8	260,0	205,4	205,6	205,8	206,7	79,48

Tabulka E.16: Statistické veličiny vypočtené z časů potřebných k vykonání výsledných kódů napsaných v jazyce Object Pascal, přeložených překladačem FreePascal a realizujících úlohu č. 3 s použitím datového typu Single.

Optimalizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
O0	50	1222,5	1267,1	1226,4	1222,6	1222,7	1223,2	10,9	0,89
O3	50	668,3	704,9	673,4	668,6	668,9	671,4	9,7	1,43
O4	50	671,0	706,2	675,0	671,2	671,4	673,4	9,2	1,36

F Úloha č. 3 – doby překladu

Tabulka F.1: Statistické veličiny vypočtené z časů nutných k překladu kódů napsaných v jazyce C++, přeložených překladačem ICC a realizujících úlohu č. 3.

Optimalizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
Od	20	3356,4	3711,8	3404,1	3364,0	3369,6	3388,0	96,7	2,84
O1	20	4825,7	5164,1	4893,0	4837,3	4850,9	4926,1	90,5	1,85
Ov	20	7857,8	8899,0	7985,0	7875,6	7896,4	7947,5	256,0	3,21

Tabulka F.2: Statistické veličiny vypočtené z časů nutných k překladu kódů napsaných v jazyce C++, přeložených překladačem MSVC a realizujících úlohu č. 3.

Optimalizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
Od	20	2432,9	2999,2	2484,8	2441,4	2445,7	2474,0	123,9	4,99
O2	20	3329,9	3597,0	3374,3	3343,0	3352,4	3361,9	63,8	1,89
Ov	20	3337,6	3666,1	3379,0	3340,8	3349,9	3375,1	76,3	2,26

Tabulka F.3: Statistické veličiny vypočtené z časů nutných k překladu kódů napsaných v jazyce C++, přeložených překladačem G++ a realizujících úlohu č. 3.

Optimalizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
O0	20	3177,4	3251,9	3193,3	3184,1	3187,8	3189,4	19,7	0,62
O3	20	3803,4	3921,8	3823,3	3807,4	3810,8	3820,7	29,9	0,78
Ov	20	3808,0	4506,6	3872,3	3812,7	3815,4	3853,7	162,7	4,20

Tabulka F.4: Statistické veličiny vypočtené z časů nutných k překladu kódů napsaných v jazyce C, přeložených překladačem ICC a realizujících úlohu č. 3.

Optimalizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
Od	30	1034,1	1249,9	1057,5	1039,3	1045,7	1055,6	42,1	3,98
O1	30	1144,3	1215,3	1158,7	1147,6	1151,9	1166,6	17,1	1,47
Ov	30	1444,8	1728,4	1470,6	1447,7	1451,3	1470,9	53,9	3,66

Tabulka F.5: Statistické veličiny vypočtené z časů nutných k překladu kódů napsaných v jazyce C, přeložených překladačem MSVC a realizujících úlohu č. 3.

Optimalizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
Od	30	572,5	924,8	597,3	573,6	576,7	582,2	73,7	12,35
O2	30	708,4	937,7	724,8	714,1	714,9	721,1	41,0	5,65
Ov	30	712,3	965,7	730,8	715,4	719,6	725,7	45,6	6,24

Tabulka F.6: Statistické veličiny vypočtené z časů nutných k překladu kódů napsaných v jazyce C, přeložených překladačem GCC a realizujících úlohu č. 3.

Optimalizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
O0	30	1061,0	1342,3	1077,5	1063,4	1064,5	1066,7	52,0	4,82
O3	30	1317,1	1489,9	1331,3	1318,7	1321,0	1324,6	32,9	2,47
Ov	30	1326,5	1697,0	1343,8	1327,6	1328,9	1331,9	67,2	5,00

Tabulka F.7: Statistické veličiny vypočtené z časů nutných k překladu kódů napsaných v jazyce Java, přeložených Java překladači a realizujících úlohu č. 3.

JVM	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
HotSpot	30	842,8	1019,4	871,9	858,7	860,7	875,1	38,4	4,40
OpenJ9	30	1135,9	1377,2	1165,5	1149,1	1152,3	1159,4	44,8	3,84

Tabulka F.8: Statistické veličiny vypočtené z časů nutných k překladu kódů napsaných v jazyce Object Pascal, přeložených překladačem FreePascal a realizujících úlohu č. 3.

Optima-lizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
O0	30	248,8	369,3	258,7	253,4	255,6	256,6	21,0	8,13
O3	30	252,8	275,5	261,2	256,4	260,0	265,8	5,7	2,19
O4	30	253,5	301,7	260,8	256,1	258,4	262,2	8,9	3,40

G Úloha č. 4 – doby běhu

Tabulka G.1: Statistické veličiny vypočtené z časů potřebných k vykonání výsledných kódů napsaných v jazyce C++, přeložených překladačem ICC a realizujících úlohu č. 4.

Optimalizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
Od	20	4755,4	4862,7	4776,2	4755,9	4764,9	4768,3	32,4	0,68
Os	20	863,3	902,5	872,6	863,5	864,8	884,4	13,9	1,59
O1	20	943,8	948,8	945,2	944,1	944,7	946,1	1,3	0,14
O2	20	373,8	387,0	375,2	374,1	374,2	374,5	3,1	0,82
O3	20	370,7	390,7	372,5	371,0	371,1	372,2	4,4	1,18
Ov	20	370,9	375,6	371,7	371,2	371,6	371,9	1,0	0,27

Tabulka G.2: Statistické veličiny vypočtené z časů potřebných k vykonání výsledných kódů napsaných v jazyce C++, přeložených překladačem MSVC a realizujících úlohu č. 4.

Optimalizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
Od	20	3248,4	3315,1	3255,2	3248,8	3249,3	3250,1	18,2	0,56
O1	20	423,4	432,4	424,6	423,9	424,2	424,4	1,9	0,44
O2	20	418,7	419,7	419,0	418,8	419,0	419,1	0,2	0,05
Ov	20	418,5	430,7	419,7	419,0	419,2	419,2	2,6	0,62

Tabulka G.3: Statistické veličiny vypočtené z časů potřebných k vykonání výsledných kódů napsaných v jazyce C++, přeložených překladačem GCC a realizujících úlohu č. 4.

Optimalizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
O0	20	3665,6	3759,2	3684,4	3666,6	3667,1	3669,6	35,4	0,96
Os	20	431,5	432,7	431,9	431,7	431,9	432,1	0,3	0,07
O1	20	416,7	420,8	417,4	417,0	417,1	417,4	0,9	0,22
O2	20	360,7	374,0	363,3	361,9	362,7	363,2	3,2	0,88
O3	20	362,9	368,6	364,6	363,4	363,6	366,1	1,7	0,47
Ofast	20	365,0	369,3	365,9	365,2	365,4	365,9	1,2	0,34
Ov	20	363,3	366,7	363,9	363,4	363,8	364,0	0,8	0,23

Tabulka G.4: Statistické veličiny vypočtené z časů potřebných k vykonání výsledných kódů napsaných v jazyce C, přeložených překladačem ICC a realizujících úlohu č. 4.

Optimalizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
Od	20	1138,0	1190,8	1148,5	1138,2	1138,6	1156,6	17,3	1,51
Os	20	386,9	395,9	388,0	387,3	387,6	387,8	1,9	0,49
O1	20	358,5	378,7	359,9	358,6	358,8	359,4	4,4	1,23
O2	20	365,1	401,8	369,2	365,7	366,5	369,0	8,1	2,20
O3	20	366,0	388,4	368,8	367,4	367,7	368,5	4,7	1,27
Ov	20	364,5	385,1	368,3	365,6	366,2	367,8	5,2	1,4

Tabulka G.5: Statistické veličiny vypočtené z časů potřebných k vykonání výsledných kódů napsaných v jazyce C, přeložených překladačem MSVC a realizujících úlohu č. 4.

Optimalizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
Od	20	1123,8	1213,9	1133,8	1123,9	1124,2	1126,8	23,7	2,09
O1	20	421,4	423,0	421,9	421,7	421,9	422,0	0,4	0,09
O2	20	413,8	434,9	415,0	413,9	414,0	414,2	4,7	1,12
Ov	20	446,1	454,2	451,2	450,2	451,3	452,2	1,8	0,39

Tabulka G.6: Statistické veličiny vypočtené z časů potřebných k vykonání výsledných kódů napsaných v jazyce C, přeložených překladačem GCC a realizujících úlohu č. 4.

Optimalizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
O0	20	1149,7	1166,7	1152,0	1150,0	1150,4	1152,2	4,0	0,34
Os	20	417,8	424,9	418,7	417,9	418,0	418,2	1,9	0,46
O1	20	405,9	426,8	407,7	406,2	406,3	407,8	4,6	1,12
O2	20	309,5	330,8	311,4	309,9	310,4	310,7	4,6	1,48
O3	20	270,1	272,1	270,7	270,3	270,4	270,9	0,6	0,20
Ofast	20	266,2	267,8	266,7	266,4	266,6	266,9	0,4	0,15
Ov	20	265,2	283,7	266,8	265,5	265,9	266,1	4,0	1,50

Tabulka G.7: Statistické veličiny vypočtené z časů potřebných k vykonání výsledných kódů napsaných v jazyce Java, přeložených Java překladači a realizujících úlohu č. 4.

JVM	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
HotSpot	20	891,8	1139,2	1 118,4	1128,8	1130,6	1132,6	53,7	4,80
OpenJ9	20	911,5	1271,5	952,8	913,4	914,6	915,9	98,5	10,34

Tabulka G.8: Statistické veličiny vypočtené z časů potřebných k vykonání výsledných kódů napsaných v jazyce Object Pascal, přeložených překladačem GCC a realizujících úlohu č. 4.

Optimalizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
O0	20	1010,4	1014,8	1011,5	1010,9	1011,1	1011,6	1,2	0,11
Os	20	1005,9	1014,3	1008,1	1007,3	1007,9	1008,7	1,8	0,17
O1	20	1016,4	1022,8	1018,7	1018,2	1018,5	1019,2	1,4	0,13
O2	20	463,6	464,4	463,9	463,7	463,9	464,0	0,2	0,05
O3	20	463,5	464,9	464,0	463,7	463,9	464,3	0,4	0,09
O4	20	480,5	482,3	481,0	480,6	480,9	481,2	0,4	0,09
Ov	20	463,2	470,7	463,9	463,3	463,5	463,8	1,6	0,35

H Úloha č. 4 – doby překladu

Tabulka H.1: Statistické veličiny vypočtené z časů nutných k překladu kódů napsaných v jazyce C++, přeložených překladačem ICC a realizujících úlohu č. 4.

Optimalizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
Od	15	2790,7	3052,7	2857,8	2799,2	2810,7	2943,2	95,0	3,33
O1	15	4150,3	4654,0	4239,2	4157,9	4185,8	4237,3	141,0	3,33
O3	15	6491,6	7045,5	6599,7	6503,3	6517,2	6565,6	180,0	2,73

Tabulka H.2: Statistické veličiny vypočtené z časů nutných k překladu kódů napsaných v jazyce C++, přeložených překladačem MSVC a realizujících úlohu č. 4.

Optimalizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
Od	15	2012,6	2194,7	2067,7	2022,9	2027,3	2106,6	61,9	2,99
O2	15	2713,2	3040,8	2755,8	2719,4	2731,6	2749,3	81,2	2,95
Ov	15	2707,5	3034,9	2764,1	2713,4	2727,1	2750,3	99,9	3,61

Tabulka H.3: Statistické veličiny vypočtené z časů nutných k překladu kódů napsaných v jazyce C++, přeložených překladačem G++ a realizujících úlohu č. 4.

Optimalizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
O0	15	2696,1	2946,3	2721,4	2700,3	2704,3	2709,1	62,7	2,30
O2	15	3068,3	3472,0	3143,0	3071,3	3082,0	3119,6	130,9	4,16
Ov	15	3132,4	3454,4	3168,3	3135,4	3138,0	3149,9	83,0	2,62

Tabulka H.4: Statistické veličiny vypočtené z časů nutných k překladu kódů napsaných v jazyce C, přeložených překladačem ICC a realizujících úlohu č. 4.

Optimalizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
Od	30	924,4	1021,2	949,0	930,1	933,9	964,1	28,2	2,97
O1	30	1003,8	1301,5	1020,6	1006,4	1009,8	1013,5	53,4	5,23
Ov	30	1190,4	1485,7	1206,3	1193,1	1195,9	1198,5	53,0	4,39

Tabulka H.5: Statistické veličiny vypočtené z časů nutných k překladu kódů napsaných v jazyce C, přeložených překladačem MSVC a realizujících úlohu č. 4.

Optimalizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
Od	30	511,3	603,0	532,4	517,6	521,9	548,9	21,3	4,00
O2	30	581,1	617,4	590,4	583,8	584,8	591,7	11,7	1,97
Ov	30	578,8	733,7	613,1	585,4	594,9	627,1	39,6	6,46

Tabulka H.6: Statistické veličiny vypočtené z časů nutných k překladu kódů napsaných v jazyce C, přeložených překladačem GCC a realizujících úlohu č. 4.

Optimalizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
O0	30	1148,5	1237,0	1161,1	1150,6	1151,8	1153,5	24,5	2,11
O3	30	1149,3	1441,0	1175,2	1151,0	1152,1	1158,8	59,2	5,04
Ov	30	1146,0	1248,6	1154,0	1147,7	1148,6	1149,8	20,6	1,79

Tabulka H.7: Statistické veličiny vypočtené z časů nutných k překladu kódů napsaných v jazyce Java, přeložených Java překladači a realizujících úlohu č. 4.

JVM	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
HotSpot	15	818,6	845,0	831,4	826,3	830,6	835,8	7,8	0,94
OpenJ9	15	1115,2	1430,6	1143,1	1118,4	1122,4	1128,0	79,7	6,97

Tabulka H.8: Statistické veličiny vypočtené z časů nutných k překladu kódů napsaných v jazyce Object Pascal, přeložených překladačem FreePascal a realizujících úlohu č. 4.

Optima-lizace	Počet měření	Min. hodnota (ms)	Max. hodnota (ms)	Aritmetický průměr (ms)	Dolní kvartil (ms)	Medián (ms)	Horní kvartil (ms)	Směrodat. odchylka (ms)	Variační koeficient (%)
O0	30	238,7	245,5	240,7	239,7	240,3	241,6	1,7	0,69
O2	30	238,5	252,0	243,0	241,3	242,8	244,6	3,1	1,26
Ov	30	240,0	307,9	246,9	241,4	242,9	244,5	13,3	5,38