Oponentský posudek BP

Objektivní analýza výkonu překladačů vybraných programovacích jazyků

Jan Pizúr

První část práce je dobře zpracovaný teoretický úvod do problematiky analýzy generovaného kódu. Nicméně by se u C++ slušelo doplnit možnost optimalizace kódu pomocí šablon a zmínit stav std::execution::par_unseq, vzhledem k silné orientaci práce na autovektorizaci.

Student porovnal překladače na platformě MS Windows. Konkrétně MingW, Microsoft Visual C++ (MSVC), Intel C++ (ICC), FreePascal, HotSpot a OpenJ9. Proč chybí projekt LLVM, který používá jak CLang, tak současné Delphi? Student srovnal dobu překladu, běhu a spotřebu paměti za běhu – bez optimalizace a se dvěma různými stupni optimalizace. Vzhledem k použití nástroje VTune, viz dále, mohl uvést i hodnoty hw počítadel, které popsal v teoretickém úvodu.

V jazycích C, C++, Java a Pascal student implementoval násobení matic (ve třech verzích), Gaussovu eliminační metodu (bez pivotace) pro datové typy float a double, a zpětné vyhledávání v grafu. Zdrojové kódy si v jednotlivých případech dostatečně odpovídají. To sice na jednu stranu umožnilo objektivní srovnání, ale na druhou stranu tím student nedosáhnul na maximální možnou optimalizaci, která je prakticky možná v C a C++, teoreticky i v Pascalu. Například v metodě MatrixMult::performMultiplication() použil 3x ternární operátor namísto toho, aby si paměť zarovnal na násobky SIMD registru (efekt alokátoru ICC). Tím by odstranil ternární operátor ve všech případech, a umožnil autovektorizaci i v případě MSVC. U Gaussovy eliminace de-facto postupoval stejně, kdy paměť matice A mohl rozšířit o vektor pravé strany. Namísto toho použil dvě pole, čímž ovšem znemožnil plné využití vektorových instrukcí. K indexaci polí v cyklech for použil datový typ int, namísto size_t.

Student profiloval kód nástrojem VTune. Výsledkem byl lepší kód generovaný ICC, např. použitím klíčového slova restrict. Otázkou je, proč stejné úsilí nevěnoval i MSVC. Ačkoliv se student snažil o autovektorizaci, nepoužil nástroj Threading Adviser pro SIMD analýzu.

Před formulací svých závěrů měl student nejprve stanovit hypotézu, podle ní sestavit test, a následně hypotézu konfrontovat s naměřenými hodnotami. Např. mohl stanovit, že použití pole je rychlejší než generiky kvůli type-erasure (viz Obr. 8.5). Naměřené hodnoty by pak hypotézu potvrdily, nebo daly důvod k další analýze. Jedině tak by bylo možné rozhodnout, zda byla chyba v implementaci, testu či neplatné hypotéze. Takto student pouze předložil empiricky získané hodnoty a jejich *post-hoc* zdůvodnění, čímž se dopustil klamu *post hoc ergo propter hoc*. Sice u každé úlohy uvádí důvody, proč ji použil, ale nedělá žádnou predikci, jak a proč by měl test dopadnout. Tím pádem nemůže ověřit, že tyto důvody jsou správné, a tudíž nemůže ověřit správnost celého testu.

Zatímco již zmíněný type-erasure je triviálně identifikovatelný důvod, např. u Gaussovy eliminace použil triviální vzor čísel 1, 9 a 100 namísto alespoň pseudonáhodných hodnot. Jak si může být jistý, že nevědomky nezpůsobil chybu, která zkreslila získané výsledky? Alespoň mohl použít např. knihovnu Eigen, aby získal referenční výsledky pro úlohy 1 – 3.

Student nijak neospravedlnil výběr testovacích úloh. V každém případě by bylo lepší, kdyby se alespoň inspiroval již standardizovanými testy, aby se vyhnul neúmyslným chybám. Např. se mohl inspirovat v literatuře DOI: 10.1145/3239235.3240499. Taková, a jí podobná, v referencích chybí. Je důležité, že vybrané úlohy student implementoval s porovnatelnou složitostí ve vybraných jazycích. Ale přece jen bych pro objektivní analýzu očekával použití některého ze SPECfp či SPECint programů, které by od sebe dokázaly dále odlišit jednotlivé C/C++ překladače.

Ve svých závěrech student komentoval zejména kód generovaný C/C++ překladači, méně častěji FreePascalem, ale bytekód Javy kód generovaný JVM nerozebírá.

Ve zkratce lze konstatovat, že implementované testy jsou zaměřeny na vyhodnocení doby překladu, doby běhu a spotřeby paměti za běhu – dle zadání. Ale již neobsahují žádnou pojistku, která by pohlídala, že naměřené hodnoty nejsou zkreslené – toto je moje hlavní výtka celé práci.

Reálně práce ukazuje, že student bakalářského studia dokáže vytvářet efektivní C++ kód, srovnatelné složitosti s kódem v Javě, který ale JVM zřetelně překonává ve výsledné době běhu a paměťové náročnosti. Na analýze a optimalizacích ICC je vidět, že student pochopil problematiku optimalizací, v rozsahu obsaženém v textu práce, a nabyté znalosti dokázal správně aplikovat.

Práci doporučuji k obhajobě a vzhledem k výše uvedeným výtkám ji hodnotím známkou

v e l m i d o b ř e

V průběhu obhajoby navrhuji, aby student zodpověděl následující otázky:

1. Proč v testu chybí LLVM, konkrétně CLang a Delphi?
2. Proč nebyly použity standardizované testy alespoň k návrhu testovacích úloh?
3. Proč nebyl analyzován bytekód Javy a kód generovaný JVM?

V Plzni dne 7. srpna 2020

Doc. Ing. Tomáš Koutný, Ph.D.
KIV-FAV-ZČU