DIVINE 4

Proč chceme mít další DIVINE?

Vladimír Štill

Masaryk University

Brno, Czech Republic

15th June 2016

Představení DIVINE 4

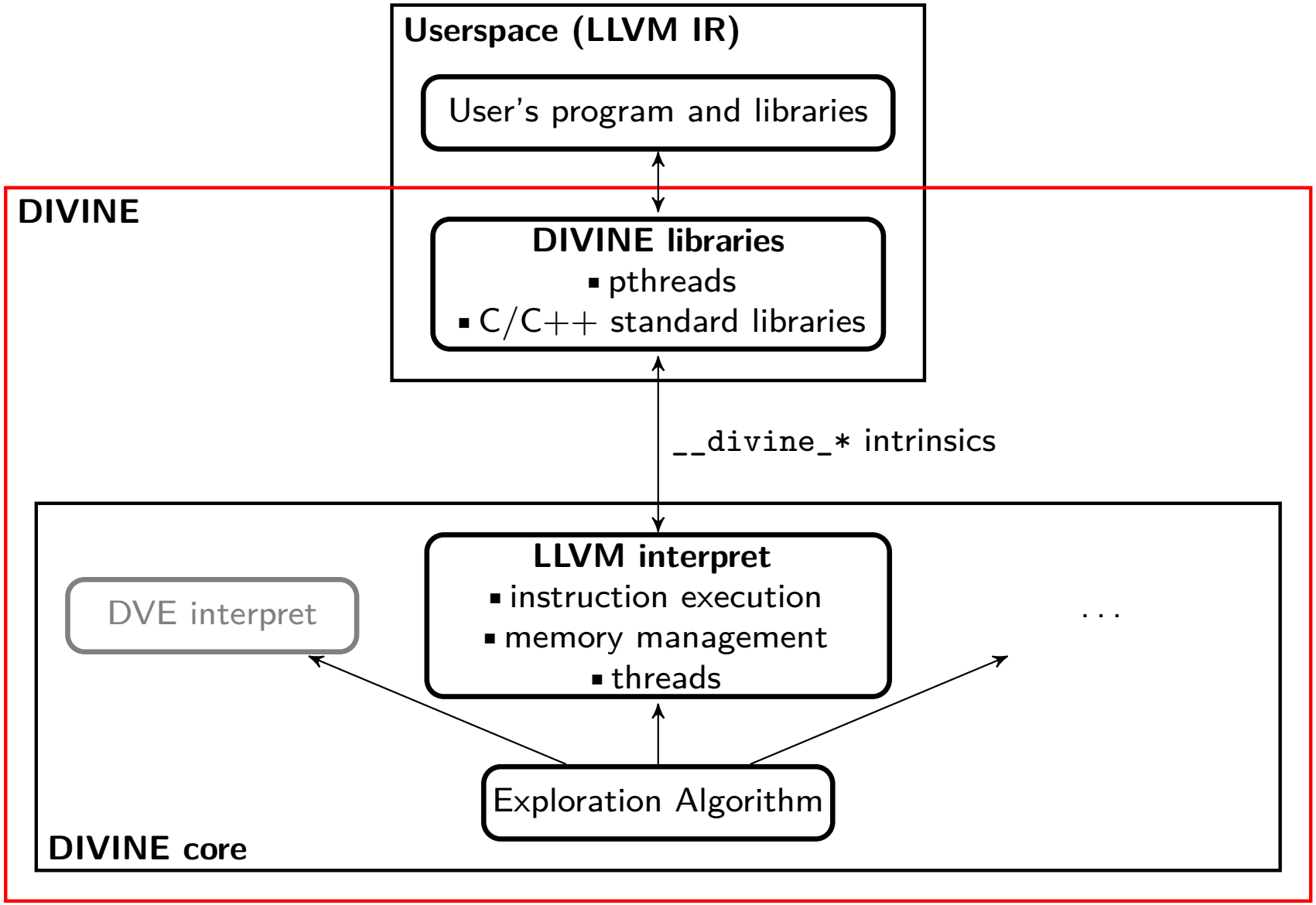
Co je DIVINE?

- nástroj na testování a verifikaci programů v C a C++
- se zaměřením na paralelní programy a detailní detekci problémů s pamětí
- využívá LLVM IR (jednodušší programovací jazyk používaný při překladu C/C++ na nativní kód)

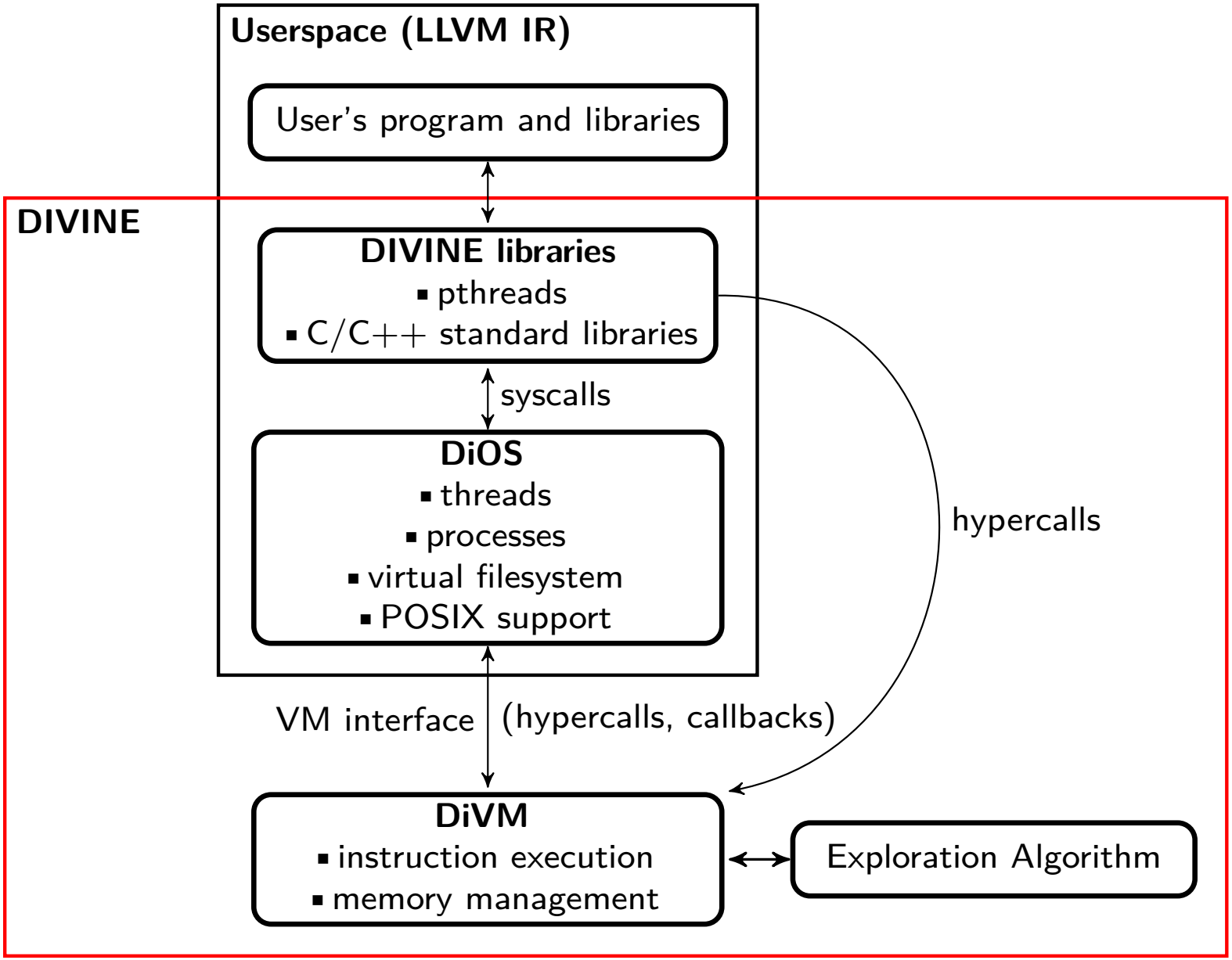
Proč nový DIVINE?

- DIVINE 3 vznikl jako první DIVINE s podporou LLVM (2012)
- zároveň zachoval podporu dalších vstupních formalismů, distribuované verifikace, LTL...
  - které DIVINE 4 zatím nemá
- pro verifikaci C/C++ postupně dosaženo limitů architektury
  - protipříklady
  - podpora procesů
  - podrobné sledování inicializovanosti paměti
  - rychlost interpretace
  - podpora různých módů plánování (asynchronní/synchronní paralelismus)

Architektura DIVINE 3



Architektura DIVINE 4



Hlavní myšlenkou DIVINE 4 je oddělení velké části kódu do DiOS

- odpovídá operačnímu systému
- DiVM (interpret) nemusí řešit plánování vláken, jen nedeterministickou volbu
- DiOS běží uvnitř DIVINE takže se snáze testuje než interpret
- DiOS lze v případě potřeby vyměnit za jiný systém s jiným plánováním (například synchronní paralelismus)
  - případně vyměnit jeho části (plánovač)
- snadněji rozšiřitelný než DiVM

DiOS: plánovač

Plánovač vláken (a procesů) je základní součástí DiOS

- samotné spouštění uživatelského kódu
- řeší prokládání vláken
- a spouštění procesů (`fork`, `exec`, případné sdílení paměti mezi procesy)
- při přerušení uživatelského kódu DiVM spustí plánovač
- uživatelský kód s DiOS komunikuje pomocí systémových volání (syscallů)

DiVM

DiVM (interpret LLVM) poskytuje rozhraní na velmi nízké úrovni (odpovídá hypervizoru)

funkce (hyperally) pro:

- správu paměti (alokace, dealokace, změna a zjištění velikosti)
- nedeterministickou volbu
- anotaci hran grafu (akceptující hrany, trace)
- oznamování cyklů a práce s pamětí (bude vysvětleno)
- ovládání control flow (bude vysvětleno)

dále definuje:

- layout rámce a volací konvence
- start programu
- způsob předávání parametrů z příkazové řádky DIVINE do programu (DiOSu)

DiVM: oznamování cyklů a práce s pamětí

DIVINE provádí redukci stavového prostoru skrýváním instrukcí, které nepřistupují k paměti viditelné jinými vlákny

- je třeba aby DIVINE vědět, kdy dochází k čtení/zápisu z/do paměti
- může detekovat přímo DiVM – ale někdy lze staticky poznat, že manipulovaná paměť je privátní
- program musí oznámit DiVM kdy přistupuje k (potenciálně) viditelné paměti
- oznámení zajistí instrumentace

DiVM: oznamování cyklů a práce s pamětí

DIVINE provádí redukci stavového prostoru skrýváním instrukcí, které nepřistupují k paměti viditelné jinými vlákny

- je třeba aby DIVINE vědět, kdy dochází k čtení/zápisu z/do paměti
- může detekovat přímo DiVM – ale někdy lze staticky poznat, že manipulovaná paměť je privátní
- program musí oznámit DiVM kdy přistupuje k (potenciálně) viditelné paměti
- oznámení zajistí instrumentace

obdobně pro cykly v control flow

- třeba detekovat kvůli terminaci hledání následníka

DiVM: oznamování cyklů a práce s pamětí

DIVINE provádí redukci stavového prostoru skrýváním instrukcí, které nepřistupují k paměti viditelné jinými vlákny

- je třeba aby DIVINE vědět, kdy dochází k čtení/zápisu z/do paměti
- může detekovat přímo DiVM – ale někdy lze staticky poznat, že manipulovaná paměť je privátní
- program musí oznámit DiVM kdy přistupuje k (potenciálně) viditelné paměti
- oznámení zajistí instrumentace

obdobně pro cykly v control flow

- třeba detekovat kvůli terminaci hledání následníka

V návaznosti na tato oznámení DIVINE provádí interrupt aktuálního výpočtu a předání řízení do plánovače.

DiVM: ovládání control flow

DiOS (případně knihovny) musí být schopny:

- vytvářet a spravovat zásobníky
- nastavovat program counter
- zakazovat interrupt (vytvářet atomické sekce)
- nastavit, která funkce řeší plánování
- nastavovat handler chyb
- nastavovat globální proměnné a konstanty (kvůli procesům)

K tomu DiVM poskytuje sadu registrů a hypercall, který je umí modifikovat a číst.

Protipříklady

Protipříklad v DIVINE 3 byl posloupností stavů

- mezi stavy může být spuštěno mnoho instrukcí (díky redukci)
- k mezivýsledkům se nedalo dostat
- samotné stavy nebylo možné přehledně reprezentovat

Protipříklady

Protipříklad v DIVINE 3 byl posloupností stavů

- mezi stavy může být spuštěno mnoho instrukcí (díky redukci)
- k mezivýsledkům se nedalo dostat
- samotné stavy nebylo možné přehledně reprezentovat

Protipříklad v DIVINE 4 je posloupností fixovaných hodnot nedeterministických voleb

- není závislý na granularitě stavů
  - lze tedy krokovat po stavech, po instrukcích, po řádcích
- k dispozici je interaktivní debugger
  - umožňuje krokovat
  - umožňuje inspekci programu v daném bodě výpočtu
  - podobně jako GDB, ale umí spolehlivě chodit i zpět
- v plánu je podpora pro generování spustitelných protipříkladů
  - pro debugování v GDB

DEMO

LLVM transformace v DIVINE 4

Postup verifikace programu v C++

1. kompilace pomocí clang API
   - využívá knihovny zabalené v DIVINE (C, C++, pthreads)
   - produkuje LLVM IR

Postup verifikace programu v C++

1. kompilace pomocí clang API
   - využívá knihovny zabalené v DIVINE (C, C++, pthreads)
   - produkuje LLVM IR

2. instrumentace a transformace
   - modifikuje LLVM IR
   - některé transformace nutné pro fungování DIVINE
   - vytváření metadat
   - redukční transformace

Postup verifikace programu v C++

1. kompilace pomocí clang API
   - využívá knihovny zabalené v DIVINE (C, C++, pthreads)
   - produkuje LLVM IR
2. instrumentace a transformace
   - modifikuje LLVM IR
   - některé transformace nutné pro fungování DIVINE
   - vytváření metadat
   - redukční transformace
3. vytváření runtime reprezentace programu
   - převod LLVM do interních struktur DIVINE pro rychlejší spouštění
   - doplňování metadat

Postup verifikace programu v C++

1. kompilace pomocí clang API
   - využívá knihovny zabalené v DIVINE (C, C++, pthreads)
   - produkuje LLVM IR
2. instrumentace a transformace
   - modifikuje LLVM IR
   - některé transformace nutné pro fungování DIVINE
   - vytváření metadat
   - redukční transformace
3. vytváření runtime reprezentace programu
   - převod LLVM do interních struktur DIVINE pro rychlejší spouštění
   - doplňování metadat
4. spuštění verifikace

Instrumentace přístupu do paměti a cyklů

- nutné pro korektnost DIVINE

přístupy do paměti se oznamují DiVM voláním hypercallu s adresou a typem přístupu

- pouze instrumentované přístupy mohou způsobit přerušení vlákna
- DiVM se může rozhodnou pokračovat bez přerušení (τ-stores redukce)

instrumentace cyklů

- instrumentují se zpětné hrany cyklů
- volání hypercallu, který může přerušit výpočet
- k přerušení dojde pokud se v průběhu generování následníka tento hypercall zavolá ze stejného místa v programu dvakrát

Statické redukce

statická escape analýza (statická τ redukce)

- k odstranění části instrumentací přístupu k paměti
- detekuje lokální proměnné, které garantovaně neopustí scope funkce
- aktuální jen jednoduchá analýza, možno rozšířit o komplikovanější escape analýzu
  - případně i s využitím points-to analýzy

$+$ odstraňování konstantních proměnných a další optimalizace z diplomky

Metadata

DiOS a knihovny potřebují metadata pro své fungování

- schopnost vyhledávat funkce podle názvu, zjišťovat počty jejich argumentů
- schopnost zjistit potřebnou velikost rámce pro funkci
- metadata pro výjimky

metadata se přidávají částečně do LLVM a částečně do runtime reprezentace

- velkou část lze napočítat staticky (argumenty, jména)
- část je závislá na tvorbě runtime reprezentace (mapování program counteru na instrukce, tabulky výjimek)

Výjimky

DIVINE je primárně zaměřen na verifikaci C++, potřebuje tedy umět výjimky

- v DIVINE 3 řešeny modifikací C++ knihovny (libc++abi) a specifickým intrinsikem pro vyhledávání landing padů a odvíjení zásobníku
- v DIVINE 4 je libc++abi nemodifikované, výjimky řešeny na úrovni metadat a unwinderu

Jak fungují výjimky? (v C++ a LLVM)

- LLVM má specifické instrukce pro volání funkce, která může zachytit výjimku: `invoke` – `landingpad` kombinace
  - `invoke` volá funkci a pokud funkce vyhodí výjimku pokračuje na příslušný `landingpad`
  - `landingpad` instrukce vrátí informace o výjimce
  - program dále výjimku ošetří nebo znovu vyhodí (`resume` instrukce)
  - program může využívat informace o type id výjimky s pomocí LLVM intrinsiků
  - landingpad odpovídá `catch` bloku nebo místu volání destruktorů
- vyhození výjimky je řešeno knihovnou jazyka (například libc++abi pro C++) spolu s knihovnou pro odvíjení zásobníku (unwinder, například libunwind)
  - tyto knihovny rovněž řeší detekci landingpadu, na který se má výjimka vypropagovat

Jak fungují výjimky? (v C++ a LLVM)

Část v knihovně jazyka by teoreticky mohla být platformě nezávislá

- prakticky ale existuje několik implementací
  - **zero-cost výjimky podle Itanium ABI** (`x86` a `x86_64` na Linuxu a Unixu)
  - zero-cost výjimky podle ARM ABI
  - výjimky pomocí `setjmp`, `longjmp` (dříve na `x86`)
  - Windows výjimky
- zero-cost výjimky používají velké množství metadat a mají drahé vyhození výjimky
  - za cenu velmi levného volání funkce, která může vyhodit výjimku
- DIVINE 4 používá Itanium ABI výjimky

Unwinder je závislý na platformě

- volací konvence, layout zásobníku, registry
- formát binárního souboru (uložení tabulek s metadaty)

Výjimky v DIVINE 4

- libc++abi v módu pro Itanium ABI unwinder je nemodifikované
- unwinder je vlastní

- libc++abi v módu pro Itanium ABI unwinder je nemodifikované
- unwinder je vlastní

DIVINE unwinder

- Itanium ABI kompatibilní unwinder
- využívá metadat o instrukcích
- jazykově nezávislý až na tabulky s metadaty pro C++ (Language-Specific Data Area)
  - lze tedy použít i pro výjimky v jiných jazycích
- exception tabulky se generují při vytváření runtime reprezentace
- samotné odvíjení zásobníku je řešeno pomocí metadat o instrukcích a nastavení rámce a program counteru v DiVM pomocí hypercallu

Závěr

DIVINE 4 je postaven na nové architektuře

- rozdělení na interpretr (DiVM), operační systém (DiOS) a uživatelský program s knihovnami
  - rozhraní (Di)VM by mělo být použitelné i pro jiné nástroje
- zjednodušení interpretru
- výrazné využívání LLVM transformací
- snadnější rozšiřování
- věrnější verifikace výjimek
- lepší protipříklady

LLVM transformace

- instrumentace přístupu do paměti a cyklů
- escape analýza (statická τ redukce)
- metadata