LLVM Transformations for Model Checking

Diplomová práce

Vladimír Štill

Masarykova univerzita

Brno, Česká republika

15. února 2016

Úvod: DIVINE

Cíl: verifikace paralelních C a C++ programů.

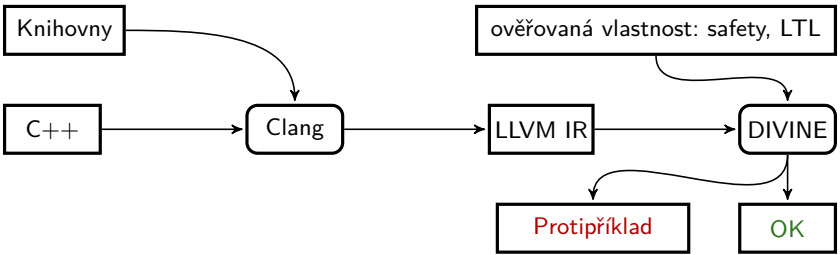
- snaha přiblížit model checking programátorům
- co možná nejširší a nejvěrnější podpora programovacích jazyků

Úvod: DIVINE

Cíl: verifikace paralelních C a C++ programů.

- snaha přiblížit model checking programátorům
- co možná nejširší a nejvěrnější podpora programovacích jazyků

- výpočetně náročné
- verifikace pomocí LLVM, využití existujících kompilátorů
- redukční strategie pro zmenšení paměťové a časové náročnosti

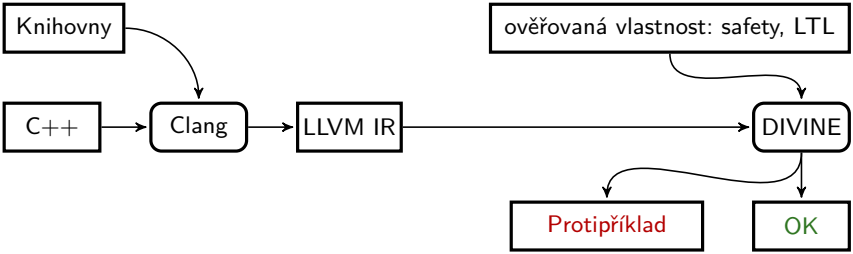


Transformace LLVM

- LLVM IR: mezijazyk používaný při překladu
- LLVM: IR + knihovny pro manipulaci s IR, generování assembleru,...
- snadná analýza, transformace

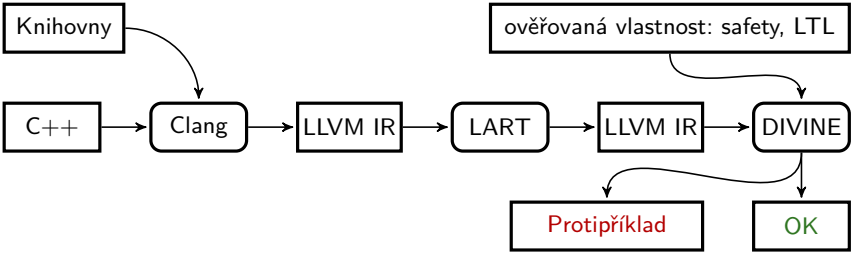
Transformace LLVM

- LLVM IR: mezijazyk používaný při překladu
- LLVM: IR + knihovny pro manipulaci s IR, generování assembleru,...
- snadná analýza, transformace



Transformace LLVM

- LLVM IR: mezijazyk používaný při překladu
- LLVM: IR + knihovny pro manipulaci s IR, generování assembleru,...
- snadná analýza, transformace



LART: nástroj pro analýzu a transformaci LLVM

- bude distribuován spolu s DIVINE
- využívá C++ API k manipulaci s LLVM IR

LART: nástroj pro analýzu a transformaci LLVM

- bude distribuován spolu s DIVINE
- využívá C++ API k manipulaci s LLVM IR

V rámci této práce rozšířen:

- analýzy a podpůrné nástroje
- **podpora pro verifikaci LLVM memory modelu**
- několik optimalizačních technik

Analýzy a podpůrné nástroje

Analýzy a podpůrné nástroje

- analýzy optimalizované na rychlost
- transformace pro usnadnění interakce s výjimkami
  - spouštění kódu na konci funkcí
  - potřebné pro mnohé další transformace
  - výjimka může způsobit ukončení funkce v místě volání jiné funkce
  - $\rightarrow$ zajištění *viditelnosti výjimek* ve funkci

Relaxované paměťové modely

Relaxované paměťové modely

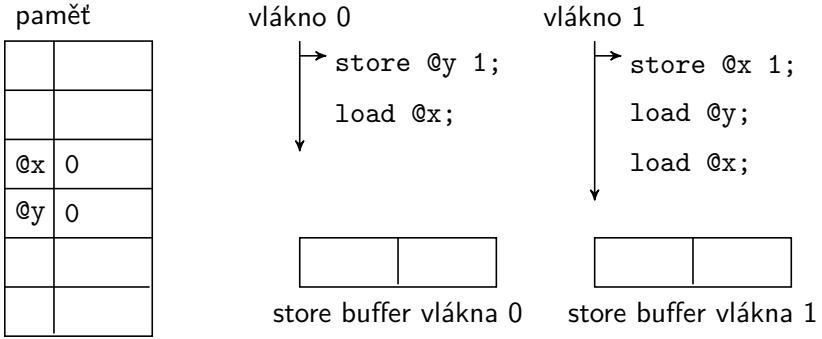
zápis do paměti není na moderních CPU okamžitě viditelný ostatním procesorům

- store buffery, cache, optimalizace cache coherence protokolu
- okamžitá viditelnost odpovídá *sekvenční konzistenci* (*Sequential Consistency*, SC)
- reálná CPU se chovají neintuitivně

Relaxované paměťové modely

zápis do paměti není na moderních CPU okamžitě viditelný ostatním procesorům

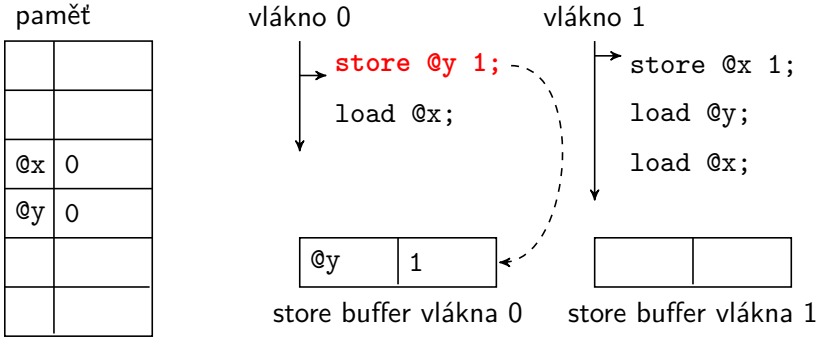
- store buffery, cache, optimalizace cache coherence protokolu
- okamžitá viditelnost odpovídá *sekvenční konzistenci* (*Sequential Consistency*, SC)
- reálná CPU se chovají neintuitivně



Relaxované paměťové modely

zápis do paměti není na moderních CPU okamžitě viditelný ostatním procesorům

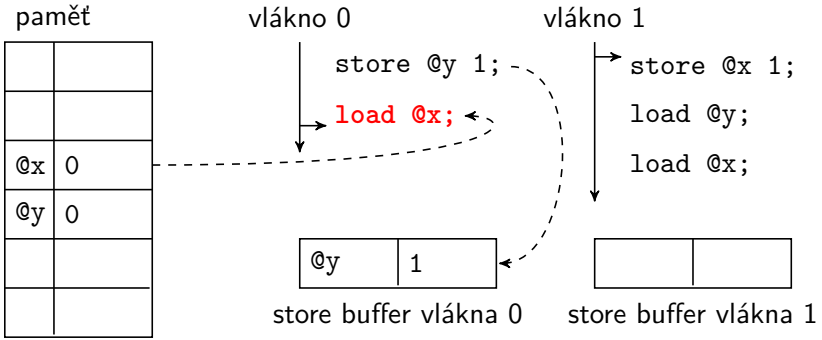
- store buffery, cache, optimalizace cache coherence protokolu
- okamžitá viditelnost odpovídá *sekvenční konzistenci* (*Sequential Consistency*, SC)
- reálná CPU se chovají neintuitivně



Relaxované paměťové modely

zápis do paměti není na moderních CPU okamžitě viditelný ostatním procesorům

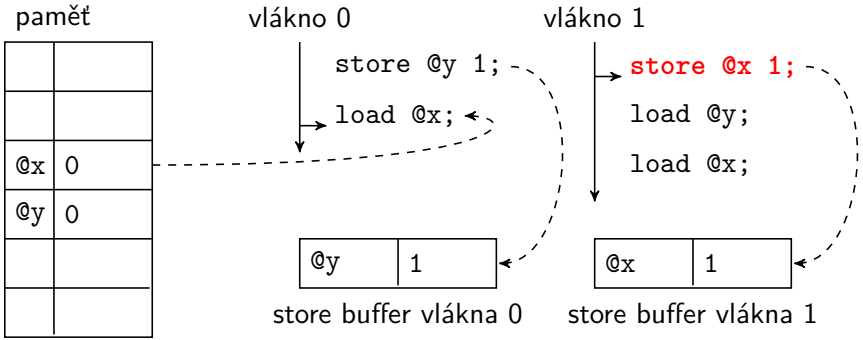
- store buffery, cache, optimalizace cache coherence protokolu
- okamžitá viditelnost odpovídá *sekvenční konzistenci* (*Sequential Consistency*, SC)
- reálná CPU se chovají neintuitivně



Relaxované paměťové modely

zápis do paměti není na moderních CPU okamžitě viditelný ostatním procesorům

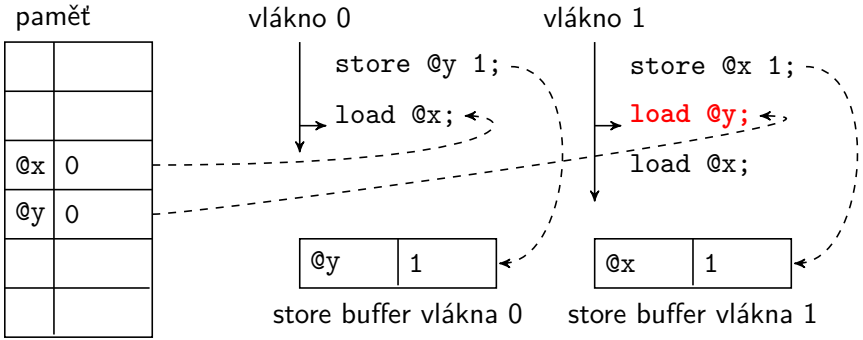
- store buffery, cache, optimalizace cache coherence protokolu
- okamžitá viditelnost odpovídá *sekvenční konzistenci* (*Sequential Consistency*, SC)
- reálná CPU se chovají neintuitivně



Relaxované paměťové modely

zápis do paměti není na moderních CPU okamžitě viditelný ostatním procesorům

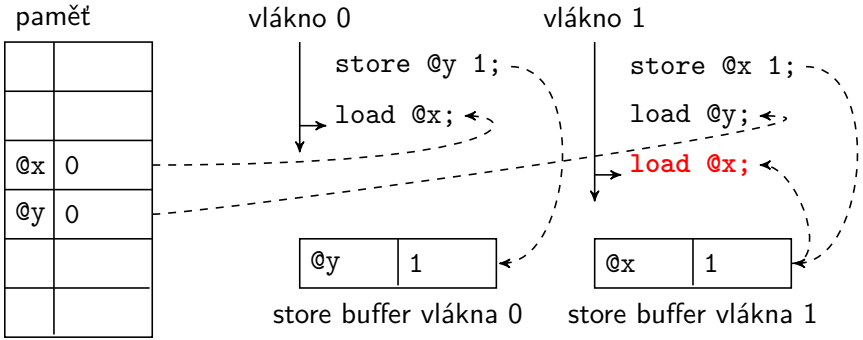
- store buffery, cache, optimalizace cache coherence protokolu
- okamžitá viditelnost odpovídá *sekvenční konzistenci* (*Sequential Consistency*, SC)
- reálná CPU se chovají neintuitivně



Relaxované paměťové modely

zápis do paměti není na moderních CPU okamžitě viditelný ostatním procesorům

- store buffery, cache, optimalizace cache coherence protokolu
- okamžitá viditelnost odpovídá *sekvenční konzistenci* (*Sequential Consistency*, SC)
- reálná CPU se chovají neintuitivně



Verifikace paměťových modelů

- verifikační nástroje často podporují jen sekvenční konzistenci
- stejně tak DIVINE

Verifikace paměťových modelů

- verifikační nástroje často podporují jen sekvenční konzistenci
- stejně tak DIVINE

dva možné přístupy k rozšíření funkcionality

modifikace interpretru

- specifické pro DIVINE
- vyžaduje zásah do složitého interpretru LLVM
- velmi náročné na ladění při vývoji
- nevyžaduje nic navíc od uživatele, jen zapnout při verifikaci modelu

Verifikace paměťových modelů

- verifikační nástroje často podporují jen sekvenční konzistenci
- stejně tak DIVINE

dva možné přístupy k rozšíření funkcionality

modifikace interpretru

- specifické pro DIVINE
- vyžaduje zásah do složitého interpretru LLVM
- velmi náročné na ladění při vývoji
- nevyžaduje nic navíc od uživatele, jen zapnout při verifikaci modelu

modifikace vstupního programu

- na úrovni C++, nebo **LLVM**
- na úrovni LLVM výrazně jednodušší
- lze modifikovat pro jiné nástroje než DIVINE
- nevyžaduje modifikaci programu uživatelem, jen zapnout při překladu

LLVM transformace pro paměťové modely

- přidává podporu LLVM paměťového modelu do programu
- téměř úplné pokrytí paměťového modelu C++11
- paměťový model definuje atomické instrukce (compare-and-swap, read-modify-write)
- simulace pomocí store bufferů rozšířených o informace o instrukci

LLVM transformace pro paměťové modely

- přidává podporu LLVM paměťového modelu do programu
- téměř úplné pokrytí paměťového modelu C++11
- paměťový model definuje atomické instrukce (compare-and-swap, read-modify-write)
- simulace pomocí store bufferů rozšířených o informace o instrukci

- transformace je parametrizovatelná
  - jaké garance paměťový model dává (reálné paměťové modely dávají často větší garance než LLVM)
  - jak velký má být store buffer

Princip transformace

- každý zápis, čtení, atomická instrukce a paměťová bariéra transformovány
  - zápis proběhne do store bufferu
  - čtení kontroluje, jestli je v lokálním store bufferu nová hodnota
- přidáno vlákno, které vylévá store buffer

Princip transformace

- každý zápis, čtení, atomická instrukce a paměťová bariéra transformovány
  - zápis proběhne do store bufferu
  - čtení kontroluje, jestli je v lokálním store bufferu nová hodnota
- přidáno vlákno, které vylévá store buffer

- atomické instrukce se simulují pomocí synchronizace + atomické masky
- ostatní nahrazeny funkcemi (implementované v C++)

Princip transformace

- každý zápis, čtení, atomická instrukce a paměťová bariéra transformovány
  - zápis proběhne do store bufferu
  - čtení kontroluje, jestli je v lokálním store bufferu nová hodnota
- přidáno vlákno, které vylévá store buffer

- atomické instrukce se simulují pomocí synchronizace + atomické masky
- ostatní nahrazeny funkcemi (implementované v C++)

- nutno zabránit opožděnému zápisu po uvolnění paměti
  - vylitím store bufferu po uvolnění paměti/skončení funkce
  - dané lokace odstraněny ze store bufferu
  - vyžaduje *viditelné* výjimky

Optimalizace velikosti stavového prostoru

Transformace zhoršuje stavovou explozi $\rightarrow$ několik optimalizací

- využívá toho, že DIVINE dokáže poznat, jestli ukazatel ukazuje do thread-local paměti
- ne vždy nutné použít store buffer
  - čtení z thread-local paměti
  - zápis do paměti, která je prokazatelné thread-private po celou dobu běhu programu (staticky určeno)

Vyhodnocení: total store order (velikost stavového prostoru)

Jméno	SC	TSO			TSO: nárůst		
	-	1	2	3	1	2	3
`simple`	127	3.45 k†	5.97 k†	15.7 k†	27.1×	47×	123×
`peterson`	703	21.8 k†	53.4 k†	55.7 k†	31.1×	76×	79.2×
`fifo`	791	14.9 k	35.9 k	48.8 k	18.8×	45.3×	61.7×
`fifo-at`	717	39.5 k	167 k	497 k	55.1×	232×	693×
`fifo-bug`	1.61 k†	11.3 k†	44.2 k†	68.7 k†	7.01×	27.4×	42.6×
`hs-2-1-0`	891 k	250 M	–	–	281×	–	–

†DIVINE nalezl v modelu chybu

Vyhodnocení: LLVM memory model (velikost stavového prostoru)

Jméno	SC	LLVM			LLVM: nárůst		
	-	1	2	3	1	2	3
`simple`	127	3.52 k†	8.07 k†	23.6 k†	27.7×	63.6×	186×
`peterson`	703	22 k†	56.3 k†	69.8 k†	31.3×	80.1×	99.3×
`fifo`	791	18.3 k	15.6 k†	23 k†	23.1×	19.8×	29.1×
`fifo-at`	717	53.5 k	256 k	1.07 M	74.6×	357×	1489×
`fifo-bug`	1.61 k†	12.1 k†	14.1 k†	21.1 k†	7.53×	8.78×	13.1×
`hs-2-1-0`	891 k	251 M	–	–	282×	–	–

†DIVINE nalezl v modelu chybu

Vyhodnocení: další experimenty (po odevzdání)

Jméno	SC	TSO				LLVM	
		1	2	4	8	2	4
`hs-2c`	2.08 k	60.3 k	76.2 k	102 k	134 k	9.39 M	–
`hs-2c+resize`	26.7 k	901 k	1.15 M	1.56 M	1.96 M	110 M	–
`hs-2d`	2.35 k	70.1 k	88.2 k	118 k	156 k	10.5 M	–
`hs-2d+2c`	7.86 k	351 k	469 k	663 k	948 k	69.5 M	–
`fifo`	10 k	204 k	326 k	480 k	593 k	491 k†	538 k†
`fifo-at`	9.33 k	216 k	472 k	1.26 M	1.65 M	10.1 M	240 M
`blockring`	2.37 k	20.1 k	27.6 k	36.3 k	40 k	10.4 k†	11 k†
`blocklink`	1.37 k†	5.38 k†	4.87 k†	3.45 k†	3.49 k†	4.48 k†	10.8 k†
`dynarray`	95 k	2.07 M	2.7 M	3.08 M	3.13 M	28.9 M	–
`vector`	162 k	8.15 M	8.95 M	10.3 M	10.6 M	333 M	–

†DIVINE nalezl v modelu chybu

Optimalizace pro verifikaci

Optimalizace pro verifikaci

- využití optimalizací pro zmenšení stavového prostoru, nebo paměťových nároků
- nesmí změnit platnost verifikovatelných vlastností
- nesmí zvětšit stavový prostor

Optimalizace pro verifikaci

- využití optimalizací pro zmenšení stavového prostoru, nebo paměťových nároků
- nesmí změnit platnost verifikovatelných vlastností
- nesmí zvětšit stavový prostor

- $\rightarrow$ generické optimalizace kompilátoru nelze použít
- navrženo několik optimalizací

Optimalizace pro verifikaci

- využití optimalizací pro zmenšení stavového prostoru, nebo paměťových nároků
- nesmí změnit platnost verifikovatelných vlastností
- nesmí zvětšit stavový prostor

- $\rightarrow$ generické optimalizace kompilátoru nelze použít
- navrženo několik optimalizací

- vesměs mají vliv na paměťovou náročnost verifikace

Optimalizace: vyhodnocení

Jméno	Počet stavů			Paměť		
	původní	opt.	úspora	původní	opt.	úspora
`fifo`	791	791	1×	380 MB	331 MB	1.15×
`fifo-tso-3`	48.8 k	42.1 k	1.16×	687 MB	510 MB	1.35×
`elevator2`	17.7 M	11.5 M	1.54×	1.24 GB	1.27 GB	0.98×
`hs-2-1-0`	891 k	875 k	1.02×	644 MB	337 MB	1.91×
`hs-2-1-0-tso-1`	250 M	184 M	1.36×	30.8 GB	18.1 GB	1.7×
`hs-2-2-2`	2.33 M	2.29 M	1.02×	1.21 GB	853 MB	1.45×

opt. = odstranění konstantních lokálních proměnných + detekce konstantních globálních proměnných

Závěr

Shrnutí

- demonstrována použitelnost LLVM transformací pro model checking v DIVINE
- podpora pro paměťové modely pomocí LLVM transformace
- optimalizace zachovávající verifikované vlastnosti

Závěr

Shrnutí

- demonstrována použitelnost LLVM transformací pro model checking v DIVINE
- podpora pro paměťové modely pomocí LLVM transformace
- optimalizace zachovávající verifikované vlastnosti

Plány do budoucna

- další snahy o redukci stavového prostoru při použití memory modelu
- využití dalších analýz k redukci stavového prostoru

Závěr

Shrnutí

- demonstrována použitelnost LLVM transformací pro model checking v DIVINE
- podpora pro paměťové modely pomocí LLVM transformace
- optimalizace zachovávající verifikované vlastnosti

Plány do budoucna

- další snahy o redukci stavového prostoru při použití memory modelu
- využití dalších analýz k redukci stavového prostoru

Děkuji za pozornost

Dotazy

Dotaz vedoucího

Bylo by možné zvýšit míru redukce stavového prostoru dosahovaného transformacemi, za předpokladu, že se verifikace programu zaměří na jeden vybraný konkrétní problém, řekněme třeba detekci deadlocku?

- ano, některé metody, například slicing, mohou fungovat lépe při omezené množině verifikovaných vlastností
- pro detekci deadlocku je však třeba zachovat implementaci synchronizace a kódu, který k ní vede

Dotazy oponenta

V obrázku 4.5 je napsáno, že řádek `foo( ptr );` je maskován, pokud byla maskována funkce volající `doSomething`. Z textu jsem však nabyl dojmu, že tento řádek není maskován za žádných okolností. Jak to tedy je?

```
1 void doSomething( int *ptr, int val ) {
2     divine::InterruptMask mask;
3     *ptr += val;
4     // release the mask only if 'mask' object owns it:
5     mask.release();
6     // masked only if caller of doSomething was masked:
7     foo( ptr );
8 }
```

Pokud není `doSomething` voláno pod maskou, pak objekt `mask` na řádku 2 vlastní masku a ta je na řádku 5 uvolněna.

Dotazy oponenta

V obrázku 4.5 je napsáno, že řádek `foo( ptr );` je maskován, pokud byla maskována funkce volající `doSomething`. Z textu jsem však nabyl dojmu, že tento řádek není maskován za žádných okolností. Jak to tedy je?

```
1  void doSomething( int *ptr, int val ) {
2      divine::InterruptMask mask;
3      *ptr += val;
4      // release the mask only if 'mask' object owns it:
5      mask.release();
6      // masked only if caller of doSomething was masked:
7      foo( ptr );
8  }
```

Pokud je `doSomething` voláno pod maskou, pak `mask` objekt nemá na tuto masku žádný vliv, a tedy řádek 7 je pod (vnější) maskou.

Dotazy oponenta

Proč se liší první číselné sloupce v tabulkách 5.5 a 5.6?

- v tabulce 5.5 představuje první sloupec neoptimalizovanou transformaci přidání memory modelů, avšak $\tau+$ redukce je v nové variantě
- v tabulce 5.6 představuje první sloupec starou verzi $\tau+$ redukce, avšak transformace je optimalizovaná
- poslední sloupec v obou tabulkách obsahuje výsledky se stejnými optimalizacemi