Zpráva – výzkumné a vývojové práce – CODASIP 2016
Výzkum a vývoj aplikací pro Codasip

Codasip Brno s.r.o.
FIT VUT v Brně

Řešení projektu se skládá v roce 2016 z několika etap a stav řešení jednotlivých etap je následující.

Etapa *Optimalizace simulátoru; implementace vývojového prostředí (zobrazení profilovacích informací)* byla rozdělena do dvou oblastí a to:

- optimalizace simulátoru a profilovacího nástroje a
- implementace vývojového prostředí, které zobrazuje profilovací informace.

Optimalizace simulátoru a profileru navazuje na etapu *Vývoj simulátoru a implementace jednoduchého vývojového prostředí*, ve které došlo k implementaci prototypů, na kterých byly otestovány vlastnosti použitých formálních modelů. Po konzultaci s pracovníky z jiných projektů, se ukázalo, že použité modely jsou vhodné, a proto mohlo dojít k optimalizaci původní implementace.

V procesu optimalizace simulátoru jsme se nejprve zaměřili na efektivnější zpracování instrukcí ze strojového jazyka. Během testování simulátoru se ukázalo, že vhodnější implementací instrukčního analyzátoru je možné snížit čas simulace až o 30%. Na základě toho zjištění došlo k vytvoření efektivnější verze instrukčního analyzátoru, jež tvoří součást vlastního simulátoru.

Jako další součást simulátoru, která vyžadovala optimalizaci, byl identifikován profiler. Profiler slouží pro nalezení úzkých míst v navrhované aplikaci a rovněž v architektuře mikroprocesoru. Aby nedocházelo k ovlivňování implementace ostatních částí projektu, byl původně profiler zapouzdřen do knihovny, která měla jasně definované rozhraní. Tato knihovna obsahovala jak architekturně závislé části (identifikace jednotlivých zdrojů, událostí atp.), tak obsahovala i architekturně nezávislé části (výpočet pokrytí instrukční sady, atp.). Využívala hojně asociativní pole (*stl::map*) a jako klíče byly použity řetězce (*std::string*). Jedna se o rychlé provedení, které však trpí řadou výkonových nedostatků. Při provádění programu je nutné profiler informovat o tom, co se děje, tedy dochází k častému volání funkcí z knihovního rozhraní. Každé zavolání znamená přepnutí kontextu, což znamená časové zpoždění. Jednotlivé zdroje a události jsou identifikovány pomocí řetězců, což znamená, že jsou během provádění simulace vytvářeny dynamicky. Vyhledávání v asociativním poli je z principu pomalé, protože i když je vnitřně implementována pomocí optimální *hash* tabulky, znamená to při každém přístupu vytváření *hash* klíče. Použití konstrukce *string* pro uchování řetězců zpomalí simulaci ještě výrazněji. Po několika testech výkonnosti se ukázalo, že je nezbytně nutné asociativní pole tam kde je to možné odstranit a nahradit je jinými strukturami a konstrukci *string* nahradit symbolickou konstantou. Ačkoli byl výsledný profiler funkční, vzhledem k zmíněným vlastnostem byl řádově pomalejší než simulátor.

Při optimalizaci došlo k rozdělení architektury profileru na dvě části. První z nich je ta, která je architekturně nezávislá. Jedná se tedy o část, která provádí výpočet statistik (viz předchozí odstavec). Tato část je převzata z původní implementace. Provádí se totiž až po ukončení běhu programu, proto zde čas potřebný na vyhodnocení není kritický. Druhá část se pak generuje pro každý procesor zvlášť. V této části se z popisu architektury procesoru vytvoří několik seznamů konstant, které slouží jako klíče do polí, které udržují informace o průběhu profilování. Tedy asociativní pole bylo nahrazeno klasickým polem, kde je klíčem číselný index. Toto nahrazení znamená výrazné zrychlení, protože odpadá zpoždění způsobené vyhledáváním v *hash* tabulce v asociativním poli a vytváření řetězců. Dále byla provedena analýza rozhraní původního profileru a funkcí, které jsou volány během simulace. Tyto funkce byly nahrazeny *makry* nebo *inline* funkcemi. Tato modifikace zajišťuje to, že během simulace nedochází ke zpoždění vlivem přepínání kontextu.

Při testování zmíněných optimalizaci se např. pro architekturu *MIPS* ukázalo, že profilování způsobí pouze sedminásobné zpoždění oproti simulátoru, což je výrazné zlepšení. Toto zpomalení je již přijatelné pro nasazení simulátoru s profilerem v praxi.

Byl vytvořen základní prototyp vývojového prostředí. Toto vývojové prostředí zobrazuje rovněž zmíněné statistické informace. Protože simulace běží v takzvané *perspektivě* platformy *Eclipse*, je i pro zobrazení profilovacích informací použita perspektiva. Byla tato perspektiva následovně optimalizována. Profilovací perspektiva je rozdělena do několika sekcí, kde každá sekce je dedikovaná jednomu řezu procesoru (viz *VLIW* mikroprocesory) a je přidána jedna sekce *globální*, která zastřešuje všechny sekce (agreguje informace ze všech řezů). Při modelování jednoduchého procesoru s jedním řezem, obsahuje globální sekce ty samé informace jako sekce pro jeden řez.

Každá sekce je rozdělena do tří základních podsekcí. První z nich zobrazuje koláčové grafy, na nichž je znázorněno pokrytí instrukční sady a programu. Druhá z nich zobrazuje tzv. *top* statistiky, tedy seznam význačných instrukcí v určitém pohledu. Je zde seznam nejčastěji používaných instrukcí, seznam instrukcí, které nejvíce přistupovaly do paměti a seznam instrukcí, které trvaly nejvíce taktů. V poslední sekci je pak seznam všech použitých instrukcí. Každá instrukce u sebe pak má několik dalších důležitých věcí, které pomáhají vývojáři v optimalizaci cílové architektury nebo programu (např. počet výpadku vyrovnávacích pamětí, atp.).

Etapa *Implementace kompilovaného simulátoru* se zaměřuje na zrychlení simulace pomocí analýzy programu, který bude na mikroprocesoru běžet. Ačkoli výhodou interpretovaného simulátoru je to, že je nezávislý na aplikaci, daní za to je jeho omezená rychlost. Každou instrukci musíme během simulace vždy načíst a poté dekódovat. Tato činnost se neustále opakuje a to i v případě, že program obsahuje smyčku (dochází k opakovanému dekódování stejných instrukcí). Je tedy zřejmé, že chceme tuto činnost eliminovat. Jak bylo řečeno, před vlastní simulací se program nejprve analyzuje. Výsledkem je seznam adres instrukcí, které se v programu vyskytují, a k těmto adresám je vygenerován potřebný kód, který by se provedl během interpretované simulace při dekódování. Tímto odstraníme neustálé dekódování stejných instrukcí. Zmíněný koncept má však problematickou část, která se skrývá v procesu analýzy. Pro větší program může být analýza zdlouhavá a může vygenerovat velké množství kódu. Toto bylo nutné mít na zřetely a proto byl navržen mechanismus pro co nejefektivnější generování výsledného kódu.

Etapa *Podpora RTL simulace a podpora profilování víceprocesorového systému na čipu* byla rozdělena do dvou oblastí a to:

- *RTL* simulace a
- profilování víceprocesorového systému.

Vytvoření simulátoru, který umožňuje provádět *RTL* simulaci (tedy takzvanou simulaci meziregistrových přenosů), je nezbytné pro realizaci mikroprocesoru v hardwaru, neboť bez provedení *RTL* simulace je nemožné odhalit úzká místa v časování architektury. Tato úzká místa by se projevila až při vyrobení a odzkoušení mikroprocesoru, ale v této fázi je nemožné již provést jejich odstranění, což má za následek nepoužití vyrobeného hardwaru a je nezbytné provést návrh nového mikroprocesoru, který daná úzká místa již nebude obsahovat. Koncept *RTL* simulátoru byl, stejně jako ostatní části simulátoru, postaven na formálním modelu, konkrétně na konečném automatu. Daný formální model nám umožňuje vygenerovat optimalizovaný *RTL* simulátor a zároveň umožňuje vygenerovat hardware, jehož funkcionalita je totožná se *RTL* simulátorem. V rámci etapy byly navrženy a implementovány metody, které umožňují transformaci popisu funkční jednotky z jazyka *ANSI C* do popisu formálního modelu (zmíněného konečného automatu),

který zachycuje časování řídící jednotky, která určuje okamžiky/takty vykonání jednotlivých částí funkční jednotky. V rámci této etapy byl navržen koncept generátoru simulátoru, který na základě formálního modelu vygeneruje jádro *RTL* simulátoru.

Během víceprocesorové simulace je nutné sledovat, který mikroprocesor komunikuje se kterým mikroprocesorem, a která data vyžadují mikroprocesory pro výměnu. Tímto sledováním můžeme odhalit úzká místa v komunikaci, např. špatný návrh sběrnic, a další neduhy návrhu, např. neustálé zneplatňování záznamu ve vyrovnávacích pamětech mikroprocesoru. Víceprocesorová simulace využívá paketů protokolu TCP/IP ke komunikaci, přičemž každý paket obsahuje identifikaci procesoru, který paket vyslal. Této informace je využito pro sbírání statistik. Vzhledem k tomu, že mikroprocesor (který obsahuje sdílený zdroj) neví, kdo všechno k tomuto zdroji může přistoupit, nelze aplikovat mechanizmus předgenerování symbolických konstant jakožto indexy do polí udržujících informace o profilování, nýbrž je nutné navrhnout optimální strukturu, která se pro tento účel použije. Při synchronní verzi víceprocesorové simulace (jeden z mikroprocesorů obsahuje generátor hodin, a tyto hodiny jsou použity i pro ostatní mikroprocesory) profiler musí odhalit vícenásobné přistupování ke zdroji, které způsobují konflikty. V rámci této etapy byly navrženy a implementovány mechanismy pro podporu profilování víceprocesorového systému na čipu.

Etapa *Návrh prototypu rekonfigurovatelného překladače C – rekonfigurace dle jazyka CodAL* se zabývala způsobem automatizace generování překladače vyššího programovacího jazyka pro konkrétní architektury mikroprocesoru. Pro programování vestavěných systémů se v současné době využívá téměř výhradně jazyk *C*, méně obvykle pak další jazyky jako *C++* nebo i *Java*. Generování překladače vyššího programovacího jazyka je tedy nezbytností.

Jako základ překladače vyššího programovacího jazyka byl zvolen systém *LLVM*, který se bude rekonfigurovat na základě cílové architektury. Přední část překladače (část téměř nezávislá na cílové architektuře) tohoto systému podporuje všechny v současnosti používané vyšší programovací jazyky. *LLVM* je poměrně nový systém s otevřeným kódem a s rychle rostoucím počtem uživatelů a především programátorů. Také je distribuován s licencí, která neomezuje komerční využití. Předpokládá se, že *LLVM* v době několika let nahradí již stárnoucí překladač *GCC*, který nyní kraluje na většině unixových systémů.

V etapě bylo navrženo rozšíření jazyka *CodAL* a také byl navržen model instrukční sady vhodný pro generování zadní části překladače (část závislá na cílové architektuře). Byl implementován program transformující model instrukční sady z jazyka *CodAL* do modelu pro generování překladače. Transformace je plně funkční pro model mikroprocesoru *MIPS32*, částečně funkční pro model *ARMv5*, pro další modely stále probíhá testování.