Programování rekonfigurovatelných systémů pomocí vyššího programovacího jazyka:

Automatická optimalizace instrukční sady na základě cílové aplikace

Projekt Lissom

Autor: Adam Husár, FIT VUT
Datum poslední úpravy: 12.12.2008
Vytvořeno jako semestrální práce do předmětu Teorie programovacích jazyků.

1 Úvod

Narůstající množství tranzistorů na čipu a nové technologie výroby integrovaných obvodů po spoustu let umožňovaly zvyšovat pracovní frekvenci procesorů. Avšak, stejně jako všechny stromy nerostou do nebe, tak se i zde objevila jistá hranice. Tou je energetická spotřeba procesorů a při taktech nad 5GHz není se současnou technologií výroby možné procesory rozumně uchladit. Tato konkrétní hranice bude jistě během několika dalších let překonána, ale problém spotřeby procesorů s námi bude až do chvíle nějakého přelomového objevu, který nabídne nové možnosti tvorby výpočetních systémů.

Pokud se podíváme na oblast přenosných zařízení a obecně spotřební elektroniky, tak se zde situace dále trochu komplikuje. U podobných přístrojů jsou nároky na spotřebu ještě striktnější, protože baterie přenosných zařízení mají omezenou kapacitu a u přístrojů napájených z elektrické sítě je často z důvodu ceny přístroje nevhodné použít u procesoru chladič.

U každého vyvíjeného zařízení také máme vždy jistý požadavek na jeho výkon, ten je obvykle obvykle určen množstvím zpracovaných dat za jistou dobu. Příkladem může být u síťového prvku jeho maximální propustnost, u dekodéru videa počet dekódovaných snímků, u grafického akcelerátoru počet operací v plovoucí řádové čárce, u databázového systému počet transakcí, u vyhledávacích systémů počet vyhledaných dat, u šifrovacího zařízení množství zašifrovaných dat, u akcelerátorů výpočtů umělé inteligence počet vykonaných operací, atd.

Například u grafických akcelerátorů se před rokem 1994 (založení spol. 3Dfx) ukázala nemožnost použít pro stejný úkol obecný procesor používaný v osobních počítačích (architektury *x*86, PowerPC, Sparc a pod.). V té době se začaly vytvářet specializované procesory pro grafické výpočty určené k použití širokou veřejností a hlavním důvodem specializace byl výkon.

Obdobně se i v dalších oblastech ukazuje nevhodnost obecných procesorů pro některé úkoly, důvody nevhodnosti jsou především cena, výkon a spotřeba. Proto vznikají nové architektury specializované na konkrétní úkol. Vhodnost nové architektury pro cílovou aplikaci však závisí pouze na vývojářích, kteří ji tvoří a na nástrojích, které mají k dispozici [Man99].

Metodologie vývoje používané před rokem 1990 se často řídily postupem *nejprve hardware a poté software* [FraPu91], [DoD85].Ukázalo se však, že je nutné vyvíjet zároveň hardware i software a zkoumat vzájemné vztahy tak, aby bylo možné ovlivnit architekturu hardwaru podle programu, který má vykonávat.

Uvedeme si požadavky, jaké na vyvíjený systém můžeme mít:

- cena,
- výkon,
- spotřeba,
- spolehlivost,
- doba odezvy,
- univerzálnost a
- doba návrhu, výroby a dodání na trh.

Téměř všechny tyto požadavky jdou proti sobě a úkolem návrháře je vybrat vhodný kompromis, tj. nalézt v množině všech možných řešení takové, které by odpovídalo stanoveným kritériím. Proces výběru řešení by se dal nazvat jako prozkoumávání hardwarových/softwarových kompromisů (trade-offs exploration, [Ku96]) a nebo také jako prozkoumávání stavového prostoru návrhu (Design Space Exploration, DSE, např. [Bai07]).

Bez současného návrhu softwaru (SW) a hardwaru (HW) je těžké prozkoumávat různé kompromisy, jako je přesun rozličných úkolů ze SW do HW (a naopak) a upravovat rozhraní mezi SW a HW. Toto téma je oblastí výzkumu a problematika zabývající se vhodným rozdělením úkolů se souhrnně nazývá HW/SW co-design. Ta obzvláště v posledním desetiletí probíhá bouřlivým vývojem.

Návrh systému obvykle probíhá tak, že máme nějaký model aplikace, často označovaný jako *zlatý model* (golden reference model) [Gaj03], nejlépe takový, který lze simulovat. Ten se postupně na základě rozhodování návrhářů upřesňuje (model refinement) a rozhoduje se, která části budou implementovány v hardwaru a které v softwaru (partitioning). Postupným upřesňováním, laděním a verifikací se tak dostaneme až do fáze, kdy je možné navrženou architekturu vyrobit v podobě polovodičového čipu a společně se softwarem použít v zařízení a to dodat na trh.

Pro některé aplikace je vhodné část softwaru úplně vypustit a veškeré úkoly implementovat v hardwaru, případně ve vyvíjeném čipu ponechat pouze jednoduchý procesor pro řízení uživatelského rozhraní a pro další výpočetně nenáročné úkoly.

Pokud v systému nepoužijeme složitější procesor, tak nám obvykle pro implementaci stejné funkčnosti stačí menší počet tranzistorů a menší pracovní takt. To se pozitivně promítne do ceny a spotřeby.

Pro aplikace, kde vyžadujeme jistou univerzálnost, tj. možnost upravit chování i poté, co byl čip vyroben, je vhodné použít jeden či více programovatelných procesorů (obvykle specializovaných).

Příkladem, kde je programovatelnost systému výhodná mohou být síťové prvky jako jsou routery a analyzátory síťového provozu. Vžijme nyní se do role společnosti, která do své síťové infrastruktury investovala nemalý obnos. Díky přechodu na protokol IPv6 je nutné provést upgrade. Programovatelným zařízením se nahraje nový program a dají se stále

používat. Avšak ty neprogramovatelná se stávají zbytečnými a jejich náhrada je pro společnost další velkou investicí.

Dalším argumentem pro programovatelná zařízení je nedokonalost nástrojů pro verifikaci hardwaru a složitost jejich používání. Pokud máme aplikaci implementovánu pouze v hardwaru a po vyrobení čipu se objeví závažná chyba, je dosti těžké tuto chybu obejít. V případě, že je čip programovatelný, máme mnohem více možností, jak se s takovou hardwarovou chybou vypořádat [Vac08].

V předchozích dvou odstavcích jsme si uvedli dvě hlavní motivace pro tvorbu programovatelných systémů. U takovýchto systémů je součástí jeden či více procesorů. Máme zde dvě možnosti: 1) použít nějaký již hotový návrh procesoru, ať už obecného (ARM, MIPS) a nebo specializovaného, např. nějaký DSP procesor, nebo 2) vytvořit specializovaný procesor přímo podle cílové aplikace a programu, který bude vykonávat.

Návrh specializovaných procesorů je časově náročná činnost a právě podporou návrhu se zabývá projekt Lissom. Projekt byl oficiálně započat v roce 2004 a jeho cílem je vytvořit vývojové prostředí, které *pomůže návrhářům vytvořit pro cílovou aplikaci v co nejkratším čase co nejvhodnější procesor*.

Úspory času se dosáhne především: 1a) co největší automatizací úkolů, které se by se jinak prováděly ručně a 1b) možností vyvíjet současně hardware i software. Dále se při hledání vhodné architektury procesoru se pomůže návrhářům tím, že se: 2a) poskytnou nástroje pro analýzu a sledování chování systému, 2b) definují metriky pro porovnávání kandidátních řešení při prozkoumávání stavového prostoru návrhu a 2c) některé kroky při prozkoumání stavového prostoru návrhu se budou provádět automaticky.

V současnosti běží po celém světě několik obdobně zaměřených projektů, avšak jediné dva se zatím dočkaly úspěšného nasazení v praxi (podle [Bai07]). Prvním je projekt LISA, jehož výsledky byly využity společností CoWare [Cow08] a druhým je projekt Target [Tar08].

Závěrem této kapitoly bych rád uvedl jednu optimistickou předpověď z roku 1990 (uvedeno v [FraPu91]).

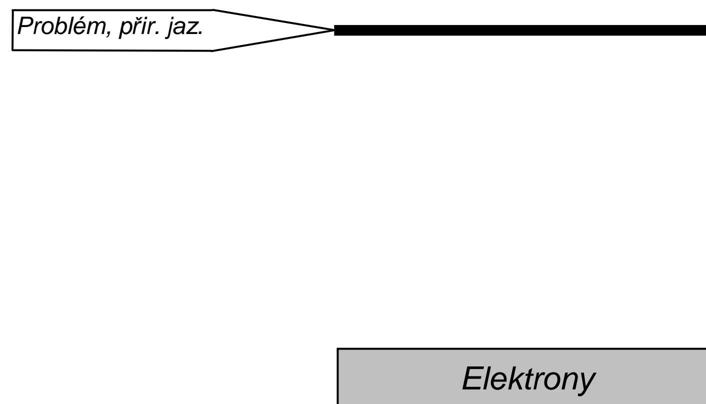
> *Ve vývojovém prostředí roku 2000 již nebudou dále existovat oddělené ostrůvky softwaru a hardwaru. Aplikace všech druhů budou dohromady spojeny pomocí softwarového lepidla, které sjednotí různé oblasti zájmu a neoddělitelně je proplete. Návrh systému bude zakotvený v tomto samotném systému a návrhář dneška se stane návrhářem zítřka.*

2 Výpočetní systém jako série překladů

Podívejme se nyní na pojem výpočetního systému z trochu jiné strany. Na začátku máme vždy nějaký *problém*, který potřebujeme vyřešit.

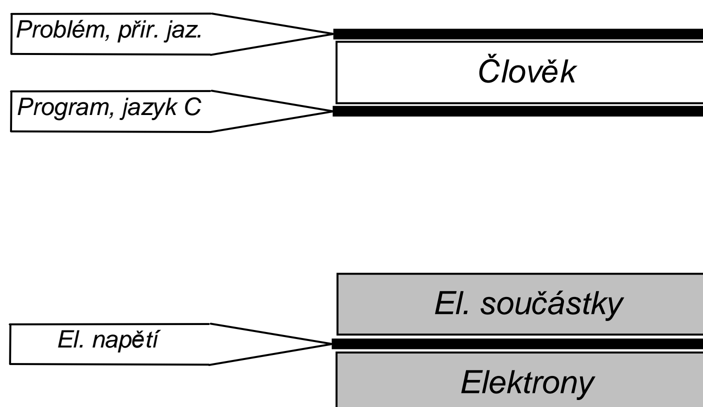
Aby byl problém řešen, potřebujeme nějakou pracovní sílu. Protože nás zde zajímají výpočetní systémy založené na softwaru a hardwaru, budou naší pracovní sílou právě *elektrony*.

Problém bývá na začátku obvykle specifikován nějakým přirozeným jazykem, a protože elektrony nerozumí ani anglickému a už vůbec ne českému jazyku, je zapotřebí jim naše zadání přeložit [Patt01].



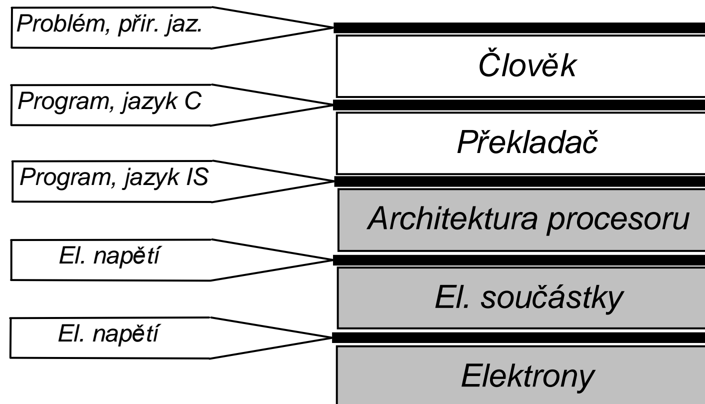
Obrázek 2.1: Propast mezi problémem specifikovaným v přirozeném jazyce a elektrony.

Lingvistická vzdálenost mezi naším jazykem a jazykem elektronů je obrovská a možných způsobů překladu je mnoho a proto si budeme si postupně prostor pro překlad omezovat. Ze strany elektronů zvolíme základní elektronické obvody (odpory, kondenzátory, polovodiče, ...). Ze strany problému, aby zadání bylo jednoznačné, zvolíme nějaký formální jazyk, zde řekněme imperativní programovací jazyk C.



Obrázek 2.2: Propast mezi problémem specifikovaným v imperativním jazyce C a elektronickými součástkami.

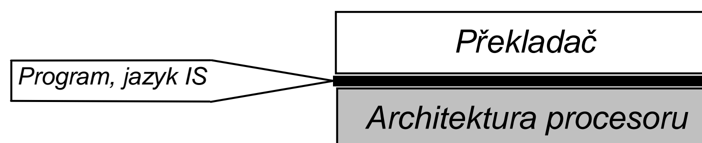
Stále je však *propast* mezi bližšími vrstvami jazyků příliš veliká, přesto se počet možností, kterými ji je lze zaplnit se podstatně zmenšil. Pokud vyjdeme z jazyka C, tak jsou možnosti v podstatě 2: 1) převést program přímo na hardware a nebo 2) použít překladač jazyka C a programovatelný procesor. Nás bude nyní zajímat především řešení s použitím procesoru.



Obrázek 2.3: Propast mezi problémem specifikovaným v imperativním jazyce C a elektronickými součástkami zaplněná pomocí překladače vyššího programovacího jazyka.

V počátcích digitální éry bylo díky malé složitosti tuto *propast* možné zaplnit ručně (tj. člověk navrhnul HW a problém převedl ručně do binární reprezentace, které hardware rozuměl). Avšak s tím, že počet tranzistorů na jednom čipu roste exponenciálně, se odpovídajícím způsobem zvyšují požadavky na složitost vyvíjeného výpočetního systému. Ruční návrh již dávno není možný a je nutné tuto činnost automatizovat.

Zde se dostáváme zpět k cíli projektu Lissom, kde chceme z kompaktního popisu instrukční sady a mikroarchitektury procesoru mimo jiné generovat překladač a popis architektury procesoru a tak co nejvíce automaticky zaplnit uvedenou *propast* mezi programovacím jazykem a elektronickými součástkami.



Obrázek 2.4: Zaplnění propasti mezi problémem zapsaným v programovacím jazyce vyšší úrovně a elektronickými součástkami.

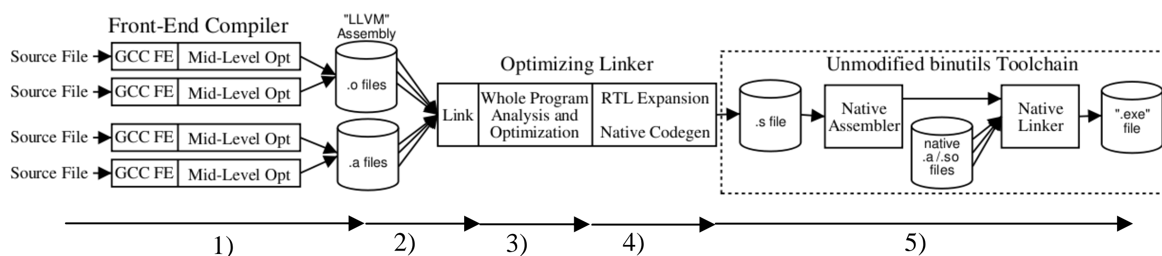
Při návrhu nového výpočetního systému často víme, jaké aplikaci bude sloužit, a proto můžeme výpočetní systém pro tuto aplikaci optimalizovat. Můžeme využít toho, že architektura a instrukční sada nejsou nastaveny napevno (jak to obvykle u již hotových architektur bývá) a můžeme *optimalizovat jak překladač, tak instrukční sadu a i architekturu procesoru jako jeden celek.*

Jasnou hranicí mezi překladačem a architekturou je jazyk instrukční sady. Ten podstatným způsobem určuje, jak bude překladač i architektura pracovat. Cílem by mohlo být nalézt optimální instrukční sadu pro danou aplikaci vzhledem k jistým stanoveným kritériím. Takovými kritérii jsou především výkon, plocha na čipu, spotřeba a univerzálnost.

Možný způsob optimalizace instrukční sady se pokusíme v následujících dvou kapitolách ukázat. Nejprve si popíšeme funkci překladače vyššího programovacího jazyka a poté naznačíme metodu optimalizace instrukční sady.

3 Postup překladu programu pomocí překladače LLVM

Projekt LLVM začal v roce 2002 na Univerzitě v Illinois (USA) s cílem vytvořit prostředí pro vývoj moderních překladačů se zaměřením na optimalizace nad celým programem [Latt02]. Tím se liší od obvyklejších překladačů, které provádějí pouze optimalizace nad jednotlivými moduly programu.



Obrázek 3.1: Architektura překladače pro optimalizace nad celým programem [LattAd03].

Překlad z jazyka vyšší úrovně do jazyka instrukční sady probíhá v následujících krocích:

1) Front-end překladače a mid-level optimalizátor přeloží program zapsaný pomocí jazyka vyšší úrovně do tzv. *univerzálního jazyka instrukční sady*, ten slouží jako mezijazyk či vnitřní reprezentace programu pro tento překladač (LLVM IR, [LLVM08a]). Jak název napovídá, jedná se o jazyk vytvořený tak, aby připomínal instrukční sady různých procesorů, také je velmi podobný klasickému tříadresnému kódu. Dále je na tento jazyk kladena podmínka, že programy musí být zapsány v tzv. Static Single Assignment podobě (SSA, např. [Much97]), tato podoba je vhodná především pro různé control- a data-flow analýzy kódu.
2) Poté se jednotlivé moduly programu spojí pomocí speciálního linkeru do celého programu.
3) Nad celým programem se provedou některé optimalizace nezávislé na cílové architektuře.
4) Nyní se vykonají různé optimalizace závislé na cílové architektuře, provede se překlad programu z jazyka univerzální instrukční sady LLVM do jazyka cílové instrukční sady a nad novou reprezentací se provedou další optimalizace. Součást vykonávající tyto překlady se obvykle nazývá jako back-end překladače.

5) Po všech uvedených krocích překladu se vygeneruje textová podoba programu v jazyce instrukční sady, ta se přeloží assemblerem do její binární podoby, poslední detaily překladu se vyřeší pomocí linkeru a vznikne nám již spustitelný program.

Celý postup prováděný překladačem LLVM by se dal shrnout jako posloupnost překladů s cílem *komprese* [CovHi03] informace tak, aby byla minimální avšak stále úplná vzhledem k jazyku cílové instrukční sady.

Pro nás je nyní nejzajímavější krok 4, který je nejvíce závislý na cílové instrukční sadě a jehož součásti budeme generovat z popisu instrukční sady. Ten si nyní podrobněji rozepíšeme.

3.1 Back-end překladače LLVM a výběr instrukcí

Back-end překladače LLVM sestává z posloupnosti průchodů. Celý seznam průchodů je uveden v příloze A, my se nyní podíváme hlavně na to, jak probíhá výběr instrukcí cílové architektury. Z hlediska plánovače průchodů LLVM jde o jeden průchod, ačkoliv sestává z 8 kroků. Jsou to tyto [LLVM08b]:

1) Sestavení prvotního orientovaného acyklického grafu (Directed Acyclic Graph, DAG) instrukcí, tento krok provede jednoduchý překlad ze vstupního kódu v LLVM IR do DAG pro výběr instrukcí.
2) Optimalizace DAG, vykonají se jednoduché optimalizace tak, aby byl DAG zjednodušen a aby byl výsledný kód efektivnější a další fáze výběru instrukcí jednodušší.
3) Legalizace datových typů, provede se transformace instrukcí nad nepodporovanými datovými typy tak, aby byl výsledek blíže cílové architektuře.
4) Optimalizace DAG, vyčistí DAG od nadbytečných operací zavedených při legalizaci datových typů.
5) Legalizace datových typů, stejný krok jako 3).
6) Optimalizace DAG, stejný krok jako 4).
7) Výběr instrukcí z DAG, spočívá v transformaci DAG s uzly ohodnocenými instrukcemi z LLVM IR do DAG, jehož uzly jsou ohodnoceny instrukcemi cílové instrukční sady.
8) Plánování a formování, vykoná se tzv. linearizace DAG, z orientovaného acyklického grafu získáme posloupnost instrukcí.

Nejdůležitějšími průchody, které následují po výběru instrukcí, je alokace registrů a plánování instrukcí.

Ve zkratce jsme si uvedli, jak probíhá výběr instrukcí. Jde o posloupnost kroků, které postupně překládají vstupní program zapsaný pomocí univerzálního jazyka instrukční sady LLVM IR tak, aby výsledkem byl ekvivalentní, pokud možno optimální program zapsaný pomocí instrukcí cílové instrukční sady.

4 Automatická optimalizace instrukční sady na základě cílové aplikace

Nyní se již dostáváme k samotnému procesu optimalizace instrukční sady. Je zde uveden pouze předběžný návrh, protože celý proces ještě není do detailů promyšlen.

Po přeložení nějaké konkrétní aplikace získáme spustitelný program a z něj si pomocí simulátoru a profileru vytipujeme tzv. žhavá místa (hot spots). To jsou části kódu, které jsou vykonávány nejčastěji a struktura takovýchto míst je určena algoritmem aplikace. Pro různé aplikace (multimédia, vyhledávání, šifrování apod.) budou mít tyto žhavá místa různou strukturu, budou se v nich objevovat různé posloupnosti instrukcí.

Optimalizace instrukční sady bude s největší pravděpodobností iterativní algoritmus. Výběr nových vhodných instrukcí architektury bude probíhat tak, že v jednotlivých iteracích prohledáme acyklický graf pro výběr instrukcí před 7. krokem výběru instrukcí (poslední krok, kdy graf stále obsahuje instrukce LLVM IR, viz kap. 3.1), konkrétně žhavá místa, podíváme se jaké vzory či posloupnosti instrukcí se v nich nejčastěji objevují a zkusíme pro některé z nich vytvořit specializované instrukce. Můžeme například zjistit, že se často opakuje dvojice vynásob a sečti a tak pro ní vytvoříme novou instrukci. Uvedený příklad je triviální, avšak nalezené vzory mohou být mnohem komplikovanější a není nemožné, že by se identifikovaly například i instrukce podobné SIMD instrukcím, či nějaké speciality jako nahraj hodnotu z paměti a inkrementuj registr a jiné kombinace.

Dá se říci, že si překladač sám vybere, které instrukce by se mu pro optimální překlad hodily, a bude je umět při překladu sám použít.

Celý proces může probíhat automaticky a iterativně se nalezne podoba instrukční sady vzhledem ke stanoveným kritériím jako je výkon a plocha na čipu.

Popsaný postup bude vhodný především pro skalární a superskalární architektury, které zřetězeně vykonávají jednotlivé instrukce a proto má u nich smysl vytvářet složitější instrukce, které mají více nezávislých efektů. Dále příklad instrukce vynásob a sečti ukazuje instrukci, která má závislé efekty, nejprve vynásobí první operand s druhým a ten pak přičte ke třetímu operandu. Na klasické architektuře by bylo zapotřebí posloupnosti dvou instrukcí, ale takto se dá zjistit, že se může vyplatit ji implementovat jako jednu a to i za cenu větší plochy na čipu. Také se dá rozpoznat, v kolika taktech by bylo vhodné takovou instrukci provést, zda v jednom a nebo ve více.

Naopak u VLIW architektur je přirozené, že se vytváří jedna VLIW instrukce (bundle) složená z více primitivních instrukcí. Pokud pro VLIW architektury zavedeme speciální instrukce s více nezávislými efekty, tak nejspíše velkého vylepšení nedosáhneme.

Otevírá se zde však další možnost. Vyjdeme z předpokladu, že jednotlivé instrukční linky VLIW procesoru mají stejné funkční jednotky (například násobička, dělička, logika pro bitový posuv, propojení na datový subsystém, …). Takováto architektura, kde jsou funkční jednotky replikované pro každý slot, je vhodná pro překladač, který má větší volnost při plánování instrukcí.

Na druhou stranu takovéto uspořádání nemusí být vhodné, protože obvykle jsou některé z replikovaných funkčních jednotek nevyužívány. Systém pro optimalizaci instrukční sady může tyto nepoužívané funkční jednotky identifikovat a odpovídajícím způsobem upravit instrukční sadu jednotlivých slotů. Příkladem architektur, které mají odlišné sloty a tedy i odlišné instrukční sady pro jednotlivé sloty jsou TI C6400 a nebo Intel Itanium IA-64. Rozdělení úkolů mezi jednotlivé sloty pak bude mít za následek menší plochu na čipu a větší vhodnost procesoru pro cílovou aplikaci.

Poznámka: Jedinou publikací, kterou jsem zatím našel a která pravděpodobně popisuje obdobný postup je [Binh97], není však veřejně dostupná a její autor v době psaní této zprávy ještě neodpověděl na email. Co se týká optimalizace počtu funkčních jednotek, tak zde se toho dá nalézt více. Dostupné publikace o optimalizaci instrukční sady mohou být například tyto: [Sap06], [Ara96] a [MoAk07].

5 Závěr

Možnost upravovat instrukční sadu procesoru pro cílovou aplikaci otevírá novou možnost optimalizovat jako jeden celek překladač, instrukční sadu a i architekturu procesoru. Proč optimalizovat překladačem pouze program, když můžeme optimalizovat i instrukční sadu?

Tím, že si překladač sám vybere, které instrukce by se mu v instrukční sadě hodily, jim bude lépe rozumět a bude schopen je sám použít. To může ve zdrojových souborech softwaru aplikace omezit množství vloženého assembleru.

Nevýhodou může být to, že vytvořená instrukční sada bude nevhodná pro jiné aplikace a pro jiný překladač. Není však obvyklé, že by se pro specializované procesory používalo více různých překladačů. Dále mohou některé později přidané optimalizace pozměnit podobu mezikódu ve fázi před výběrem instrukcí, a vzory, které v mezikódu překladač dříve viděl, už nerozpozná. Pokud bychom přidávali nové optimalizace až za proces výběru instrukcí, tak by tento problém nebyl tak vážný. Druhou možností by bylo stanovit nějakou normální formu mezikódu pro výběr instrukcí a do této formy jej vždy převést.

Při vytváření nové specializované architektury se ji snažíme vytvořit tak, aby se pro cílovou aplikaci co nejvíce hodila. Tím, že si podle potřeby můžeme upravit i hardware máme velkou volnost při výběru řešení. Jedním z procesů, které nám pomohou vybrat takové vhodné řešení, by mohl být postup uvedený v tomto dokumentu. I když se podobnými postupy mnoho lidí nezabývalo, je dle mého názoru automatická optimalizace instrukční sady uskutečnitelná.

Literatura

[Ara96] Araujo, G. a kol.: *Instruction set design and optimization for address computation in DSP architectures*. In 9th International Symposium on Systems Synthesis, 1996, dostupné na http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.38.9772 (12.12.2008).

[Bai07] Bailey, B. a kol.: *ESL Design and Verification: A Prescription for Electronic System Level Methodology*. Morgan Kauffman Publishers, 2007.

[Binh97] Binh, N. N.: *Instruction Set Optimization Algorithms for Pipelined ASIP Design,* Integrated System Design Lab, Osaka University, 1997, abstrakt dostupný na http://vlsilab.ics.es.osaka-u.ac.jp/lab_HP/english/research/thesis_in/abst97.html (12.12.2008).

[Cow08] *Internetové stránky společnosti Coware*. Dostupné na http://www.coware.com/ (10.12.2008).

[CovHi03] Coveney P.,Highfield R.: *Mezi chaosem a řádem*. Mladá Fronta, 2003.

[DoD85] *Military Standard: Defense System Software Development.* Department of Defense, Washington, 1985, dokument dostupný na http://www.barringer1.com/mil_files/DOD-2167.pdf (9.12.2008).

[FraPu91] Franke, D. W., Purvis, M. K.: *Hardware/Software CoDesign: A Perspective.* Proceedings of the 13th international conference on Software engineering*,* 1991, dostupné na http://portal.acm.org/citation.cfm?id=256664.256826 (9.12.2008).

[Gaj03] Gajski, D. D.: *System-Level Design Methodology*. University of California, Irvine, 2003, dostupné na http://www.cecs.uci.edu/~gajski/presentation/GajskYokohama.pdf (10.12.2008).

[Ku96] Kumar, S. a kol.: *The Codesign of Embedded Systems: A Unified Hardware/Software Representation*. Kluwer Academic Publishers, 1996.

[Latt02] Lattner, C. A.: LLVM: *An Infrastructure for Multi-Stage Optimization.* Master Thesis, University of Illinois at Urbaba-Champaign, 2002, dokument dostupný na http://llvm.cs.uiuc.edu/pubs/2002-12-LattnerMSThesis.pdf (9.12.2008).

[LattAd03] Lattner, C. A., Adve, V.: *Architecture for a Next-Generation GCC.* Proc. First Annual GCC Developer's Summit, 2003, dokument dostupný na http://llvm.cs.uiuc.edu/pubs/2003-05-01-GCCSummit2003.pdf (9.12.2008).

[LLVM08a] *LLVM Language Reference Manual*, The LLVM Compiler Infrastructure, 2008, dokument dostupný na http://llvm.org/docs/LangRef.html (11.12.2008).

[LLVM08b] *The LLVM Target-Independent Code Generator*, The LLVM Compiler Infrastructure, 2008, dokument dostupný na http://www.llvm.org/docs/CodeGenerator.html (12.12.2008).

[Man99] Man, H.: *System-on-Chip Design: Impact on Education and Research*. IEEE Design & Test, vol. 16, issue *3*, 1999, dostupné na http://www2.computer.org/portal/web/csdl/doi/10.1109/54.785820 (10.12.2008).

[MoAk07] Montgomery, D., Akoglu, A.: *Cryptographic Instruction Set Processor Design*. Information Security and Cryptology Conference with International Participation, Ankaram 2007, dostupné na http://www.iscturkey.org/2007/pdf/sozlu/19.pdf (12.12.2008).

[Much97] Muchnick, S. S.: *Advanced Compiler Design and Implementation*. Morgan Kauffman Publishers, 1997.

[Patt01] Patt, Y.: *Requirements, Bottlenecks, and Good Fortune: Agents for Microprocessor Evolution*. Proceedings of the IEEE, 2001, dopstupné na http://ece.iut.ac.ir/faculty/khadivi/00964437.pdf (9.12.2008).

[Sap06] Saponara, S. a kol.: *Design of Application Specific Instruction-Set Processor for Image and Video Filtering*. University of Pisa, Italy, 2006, dostupné na http://www.iss.rwth-aachen.de/4_publikationen/res_pdf/2006KammlerEUSIPCO.pdf (12.12.2008).

[Tar08] *Internetové stránky společnosti Target*. Dostupné na http://www.retarget.com/ (10.12.2008).

[Vac08] Václavík, P.: *osobní konzultace*, 2008.

Příloha A
Seznam průchodů backendu překladače LLVM pro architekturu MIPS (anglicky)

Poznámka: Příloha přidána pro úplnost, ještě není úplně dokončena.

1. Preliminary module verification

2. Dominator Tree Construction
   - Implemented in DominatorTree, subclass of DominatorTreeBase
   - Function pass
   - This file implements simple dominator construction algorithms for finding forward dominators. Postdominators are available in libanalysis, but are not included in libvmcore, because it's not needed. Forward dominators are needed to support the Verifier pass.
   - Control-flow analysis,
   - Dominators – indentify loops
   - In essence, a node A in the flowgraph dominates a node B if every path from the entry node to B includes A (SM, 171). Reflexive, transitive, antisymmetric
   - Creates dominator tree consisting of basic blocks which is then used by Module Verifier

3. Module Verifier

4. Natural Loop Construction
   - Implemented in LoopInfo,
   - This file defines the LoopInfo class that is used to identify natural loops and determine the loop depth of various nodes of the CFG. Note that natural loops may actually be several loops that share the same header node.
   - This analysis calculates the nesting structure of loops in a function. For each natural loop identified, this analysis identifies natural loops contained entirely within the loop and the basic blocks the make up the loop.
   - It can calculate on the fly various bits of information, for example:
     - whether there is a preheader for the loop
     - the number of back edges to the header
     - whether or not a particular block branches out of the loop
     - the successor blocks of the loop
     - the loop depth
     - the trip count
     - etc...

5. Canonicalize natural loops
   - Implemented in LoopSimplify
   - This pass performs several transformations to transform natural loops into a simpler form, which makes subsequent analyses and transformations simpler and more effective.
   - Loop pre-header insertion guarantees that there is a single, non-critical entry edge from outside of the loop to the loop header. This simplifies a number of analyses and transformations, such as LICM.
   - Loop exit-block insertion guarantees that all exit blocks from the loop (blocks which are outside of the loop that have predecessors inside of the loop) only have predecessors from inside of the loop (and are thus dominated by the loop header). This simplifies transformations such as store-sinking that are built into LICM.
   - This pass also guarantees that loops will have exactly one backedge.

- Note that the simplifycfg pass will clean up blocks which are split out but end up being unnecessary, so usage of this pass should not pessimize generated code.
- This pass obviously modifies the CFG, but updates loop information and dominator information.

6. Scalar Evolution Analysis
   - Implemented in SCEV
   - This file contains the implementation of the scalar evolution analysis engine, which is used primarily to analyze expressions involving induction variables in loops.
   - There are several aspects to this library. First is the representation of scalar expressions, which are represented as subclasses of the SCEV class. These classes are used to represent certain types of subexpressions that we can handle. These classes are reference counted, managed by the SCEVHandle class. We only create one SCEV of a particular shape, so pointer-comparisons for equality are legal.
   - One important aspect of the SCEV objects is that they are never cyclic, even if there is a cycle in the dataflow for an expression (ie, a PHI node). If the PHI node is one of the idioms that we can represent (e.g., a polynomial recurrence) then we represent it directly as a recurrence node, otherwise we represent it as a SCEVUnknown node.
   - In addition to being able to represent expressions of various types, we also have folders that are used to build the *canonical* representation for a particular expression. These folders are capable of using a variety of rewrite rules to simplify the expressions.
   - Once the folders are defined, we can implement the more interesting higher-level code, such as the code that recognizes PHI nodes of various types, computes the execution count of a loop, etc.

   - There are several good references for the techniques used in this analysis.
   - Chains of recurrences -- a method to expedite the evaluation of closed-form functions, Olaf Bachmann, Paul S. Wang, Eugene V. Zima
   - On computational properties of chains of recurrences, Eugene V. Zima
   - Symbolic Evaluation of Chains of Recurrences for Loop Optimization, Robert A. van Engelen
   - Efficient Symbolic Analysis for Optimizing Compilers, Robert A. van Engelen
   - Using the chains of recurrences algebra for data dependence testing and induction variable substitution, MS Thesis, Johnie Birch

   - The ScalarEvolution analysis can be used to analyze and catagorize scalar expressions in loops. It specializes in recognizing general induction variables, representing them with the abstract and opaque SCEV class. Given this analysis, trip counts of loops and other important properties can be obtained.

   - This analysis is primarily useful for induction variable substitution and strength reduction.
   - Related to Loop Optimizations

7. Loop Pass Manager
   - Implemented in LPPassManager

8. Lower Garbage Collection Instructions
   - Implemeted in LowerIntrinsics
   - This pass rewrites calls to the llvm.gcread or llvm.gcwrite intrinsics, replacing them with simple loads and stores as directed by the GCStrategy. It also performs automatic root initialization and custom intrinsic lowering.

9. Lower invoke and unwind, for unwindless code generators
   - Implemented in LowerInvoke

- This transformation is designed for use by code generators which do not yet support stack unwinding. This pass supports two models of exception handling lowering, the 'cheap' support and the 'expensive' support.
- 'Cheap' exception handling support gives the program the ability to execute any program which does not "throw an exception", by turning 'invoke' instructions into calls and by turning 'unwind' instructions into calls to abort(). If the program does dynamically use the unwind instruction, the program will print a message then abort.
- 'Expensive' exception handling support gives the full exception handling support to the program at the cost of making the 'invoke' instruction really expensive. It basically inserts setjmp/longjmp calls to emulate the exception handling as necessary.
- Because the 'expensive' support slows down programs a lot, and EH is only used for a subset of the programs, it must be specifically enabled by an option.
- Note that after this pass runs the CFG is not entirely accurate (exceptional control flow edges are not correct anymore) so only very simple things should be done after the lowerinvoke pass has run (like generation of native code).
- This should not be used as a general purpose "my LLVM-to-LLVM pass doesn't support the invoke instruction yet" lowering pass.

- Uses TargetLowering for target's jmp_buf size and alignment
  - TargetLowering::getJumpBufSize()
    1. may be different for every target: Alpha – 272B, ia64-linux – 704B, default 0
    2. what is jmp_buf size?

  - getJumpBufAlignment()
    1. Alpha – 16B, IA64 – 16B, default 0

10. Remove unreachable blocks from the CFG
    - Implemented in UnreachableBlockElim
    - This pass is an extremely simple version of the SimplifyCFG pass. Its sole job is to delete LLVM basic blocks that are not reachable from the entry node. To do this, it performs a simple depth first traversal of the CFG, then deletes any unvisited nodes.
    - Note that this pass is really a hack. In particular, the instruction selectors for various targets should just not generate code for unreachable blocks. Until LLVM has a more systematic way of defining instruction selectors, however, we cannot really expect them to handle additional complexity.
    -
    - Includes TargetInstrInfo but does not seem to use any target-specific information.

11. Optimize for code generation
    - Implemented in CodeGenPrepare
    - This pass munges the code in the input function to better prepare it for SelectionDAG-based code generation. This works around limitations in it's basic-block-at-a-time approach. It should eventually be removed.

    - Uses TargetLoweting to consult for determining transformation profitability.

    - TargetLoweting: getValueType, getTypeAction, getTypeToTransformTo, isLegalAddressingMode, getPointerTy, getTargetData, ComputeConstraintToUse, isTruncateFree, getTargetMachine
    - TargetAsmInfo: ExpandInlineAsm

12. MIPS DAG->DAG Pattern Instruction Selection
    - Implemented in MipsDAGToDAGIsel
    - MIPS specific code to select MIPS machine instructions for SelectionDAG operations.
    - Instruction selector for the MIPS target.

- Uses results from AliasAnalysis and GCModuleInfo
- Class SelectionDAGISel provides pure virtual method InstructionSelect. This method must be implemented in SelectionDAGISel subclass. For all targets, this overridden method contains only these two calls: ***DAGToDAGISel::SelectRoot(); CurDAG->RemoveDeadNodes(). Global function SelectRoot starts instruction selection, selects pending nodes from the instrtuction selection queue until no more nodes are left for the selection. Calls the TargetDAGToDAGISel::Select method on every not yet selected node. (Note: there is a little hack in here, inside of TargetDAGToDAGISel class declaration is directly included file TargetGenDAGISel.inc which further includes DAGISelHeader, so all functions inside these files are in fact inline methods of a TargetDAGToDAGISel class.)
- Inside method TargetDAGToDAGISel::Select can be specified specific selection actions for LLVM IR instructions and for all the others, TargetDAGToDAGISel::SelectCode is called.
- Hand-made instruction selection is now used mainly for instructions that define more than one result.
- Now how does the instruction selection work::...

- MipsDAGToDAGISel implements these public methods: constructor, InstructionSelect and getPassName

12. Remove unreachable machine basic blocks
    - Implemented in UnreachableMachineBlockElim

13. Live Variable Analysis
    - Implemented in LiveVariables
    - This file implements the LiveVariables analysis pass. For each machine instruction in the function, this pass calculates the set of registers that are immediately dead after the instruction (i.e., the instruction calculates the value, but it is never used) and the set of registers that are used by the instruction, but are never used after the instruction (i.e., they are killed).
    - This class computes live variables using a sparse implementation based on the machine code SSA form. This class computes live variable information for each virtual and _register allocatable_ physical register in a function. It uses the dominance properties of SSA form to efficiently compute live variables for virtual registers, and assumes that physical registers are only live within a single basic block (allowing it to do a single local analysis to resolve physical register lifetimes in each basic block). If a physical register is not register allocatable, it is not tracked. This is useful for things like the stack pointer and condition codes.

14. Eliminate PHI nodes for register allocation
    - Implemented in PNE
    - This pass eliminates machine instruction PHI nodes by inserting copy instructions. This destroys SSA information, but is the desired input for some register allocators.

15. Two-Address instruction pass
    - Implemented in TwoAddressInstructionPass
    - This file implements the TwoAddress instruction pass which is used by most register allocators. Two-Address instructions are rewritten from:

      A = B op C

    to:

A = B
A op= C

- Note that if a register allocator chooses to use this pass, that it has to be capable of handling the non-SSA nature of these rewritten virtual registers.
- It is also worth noting that the duplicate operand of the two address instruction is removed.

16. Live Interval Analysis
    - Implemented in LiveIntervals
    - This file implements the LiveInterval analysis pass which is used by the Linear Scan Register allocator. This pass linearizes the basic blocks of the function in DFS order and uses the LiveVariables pass to conservatively compute live intervals for each virtual and physical register.

17. MachineDominator Tree Construction
    - Implemeted in MachineDominatorTree
    - DominatorTree construction - This pass constructs immediate dominator information for a flow-graph based on the algorithm described in this document: A Fast Algorithm for Finding Dominators in a Flowgraph T. Lengauer & R. Tarjan, ACM TOPLAS July 1979, pgs 121-141.
    - This implements both the O(n*ack(n)) and the O(n*log(n)) versions of EVAL and LINK, but it turns out that the theoretically slower O(n*log(n)) implementation is actually faster than the "efficient" algorithm (even for large CFGs) because the constant overheads are substantially smaller.

18. Machine Natural Loop Construction
    - Implemented in MachineLoopInfo
    - This file defines the MachineLoopInfo class that is used to identify natural loops and determine the loop depth of various nodes of the CFG. Note that the loops identified may actually be several natural loops that share the same header node... not just a single natural loop.

19. Simple Register Coalescing
    - Implemented in SimpleRegisterCoalescing
    - This file implements a simple register coalescing pass that attempts to aggressively coalesce every register copy that it can.

20. Live Stack Slot Analysis
    - Implemented in LiveStacks
    - This file implements the live stack slot analysis pass. It is analogous to live interval analysis except it's analyzing liveness of stack slots rather than registers.

21. Linear Scan Register Allocator
    - Implemented in RALinScan
    - Implements a linear scan register allocator.

22. Stack Slot Coloring
    - Implemented in StackSlotColoring

23. Subregister lowering instruction pass
    - Implemented in LowerSubregsInstructionPass
    - This file defines a MachineFunction pass which runs after register allocation that turns subreg insert/extract instructions into register copies, as needed. This ensures correct codegen even if the coalescer isn't able to remove all subreg instructions.

24. Prolog/Epilog Insertion & Frame Finalization
    - Implemented in PEI
    - This pass is responsible for finalizing the functions frame layout, saving callee saved registers, and for emitting prolog & epilog code for the function.
    - This pass must be run after register allocation. After this pass is executed, it is illegal to construct MO_FrameIndex operands.

25. Control Flow Optimizer
    - Implemented in BranchFolder
    - This pass forwards branches to unconditional branches to make them branch directly to the target block. This pass often results in dead MBB's, which it then removes.
    - Note that this pass must be run after register allocation, it cannot handle SSA form.

26. Analyze Machine Code For Garbage Collection
    - Implemented in MachineCodeAnalysis
    - This file implements target- and collector-independent garbage collection infrastructure.
    - MachineCodeAnalysis identifies the GC safe points in the machine code. Roots are identified in SelectionDAGISel.

27. Label Folder
    - Implemented in DebugLabelFolder
    - This pass prunes out redundant labels. This allows a info consumer to determine if the range of two labels is empty, by seeing if the labels map to the same reduced label.

28. Mips Delay Slot Filler
    - Implemented in Filler
    - Simple pass to fills delay slots with NOPs.

29. MachineDominator Tree Construction
    - already described (18)

30. Machine Natural Loop Construction
    - already described (19)

31. Loop aligner
    - Implemented in LoopAligner
    - This file implements the pass that align loop headers to target specific alignment boundary.

32. Mips Assembly Printer
    - Implemented in MipsAsmPrinter
    - This file contains a printer that converts from our internal representation of machine-dependent LLVM code to GAS-format MIPS assembly language.

33. Delete Garbage Collector Information
    - Implemented in Deleter
    - Releases GC metadata

34. Machine Code Deleter
    - This pass deletes all of the machine code for the current function, which should happen after the function has been emitted to a .s file or to memory