VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

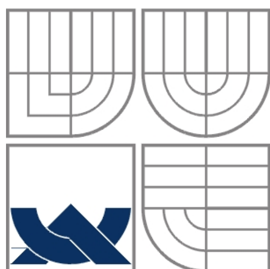
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

VIRTUÁLNÍ STROJ PARROT

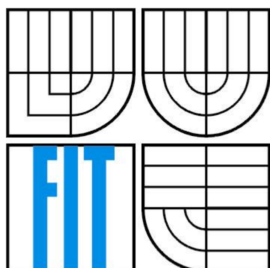
BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE Martin Mecera
AUTHOR

BRNO 2008



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

VIRTUÁLNÍ STROJ PARROT
PARROT VIRTUAL MACHINE

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE
AUTHOR

Martin Mecera

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. Zbyněk Křivka Ph.D.

BRNO 2008

Abstrakt

Práce popisuje druhy a vlastnosti moderních virtuálních strojů. V práci je vysvětlen princip činnosti každého druhu virtuálního stroje. Zvláštní úsilí je věnováno popisu virtuálních strojů pro vyšší programovací jazyky. Z těchto virtuálních strojů je vybrán Parrot. Charakteristice architektury Parrotu je věnována podstatná část bakalářské práce.

Klíčová slova

Virtuální stroj, Parrot, vyšší programovací jazyk, statický programovací jazyk, dynamický programovací jazyk.

Abstract

The thesis describes types and properties of modern virtual machines. Principles of each mentioned virtual machine are described. Substantial effort is devoted to the description of high level language virtual machines. Parrot virtual machine is depicted. Characteristics of Parrot architecture are esential part of this bachelors thesis.

Keywords

Virtual machine, Parrot, high level language, statically typed language, dynamically typed language.

Citace

Mecera Martin: Virtuální stroj Parrot. Brno, 2008, bakalářská práce, FIT VUT v Brně.

Virtuální stroj Parrot

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Zbyňka Křivky Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……………………
Martin Mecera
11. 1. 2008

Poděkování

Děkuji vedoucímu práce Ing. Zbyňku Křivkovi Ph.D. za odbornou pomoc.

© Martin Mecera, 2008.

Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů..

Obsah

Obsah 1
Úvod 2
1 Typy virtuálních strojů 3
1.1 Procesní virtuální stroj 4
1.1.1 Emulace 4
1.1.2 Optimalizace 5
1.1.3 Kompatibilita prostředí 6
1.2 Systémový virtuální stroj 7
1.3 Virtuální stroj spoluvyvíjený s hardwarem 8
1.4 Multiprocesorová virtualizace 9
2 Virtuální stroje pro jazyky vyšší úrovně 10
2.1 Virtuální instrukční sada 11
2.2 Vybrané vlastnosti vituálních strojů pro vyšší programovací jazyky 13
2.2.1 Bezpečnost 13
2.2.2 Robustnost 14
2.2.3 Síťování 15
3 Virtuální stroj Parrot 16
3.1 Interpret 16
3.2 Reprezentace objektů 17
3.3 Překladač 18
4 Závěr 21
Příloha 1: Návrh jazyka 22
Příloha 2: Gramatika jazyka 24
Literatura 27
Seznam příloh 28

Úvod

Virtuální stroje vznikají zejména proto, aby vytvářely prostředí pro komplikované procesy, které mají běžet nad větším počtem různých operačních systémů, počítačových architektur a jiných počítačových komponent. Typů virtuálních strojů je několik. Každý typ virtuálního stroje pracuje v jiném kontextu (například přímo nad hardwarem nebo operační systémem), a proto i prostředí, které nabízí, a procesy, které hostuje, jsou pro něj specifické.

V této práci popisuji nejvýznamnější druhy virtuálních strojů a uvádím jejich společné a odlišné rysy. Hlouběji se zaměřuji na virtuální stroj Parrot. Podrobně popisuji druh virtuálních strojů, jehož je Parrot členem.

Kapitola první je věnována druhům virtuálních strojů. Je popsán způsob práce každého z nich.

Kapitola druhá pojednává o vlastnostech virtuálních strojů pro vyšší programovací jazyky. V druhé kapitole jsou rozebrány některé vlastnosti instrukční sady a vlivy, které architekturu instrukční sady formují. Dále se pak věnuji důsledkům interpretace na bezpečnost, robustnost a síťový provoz.

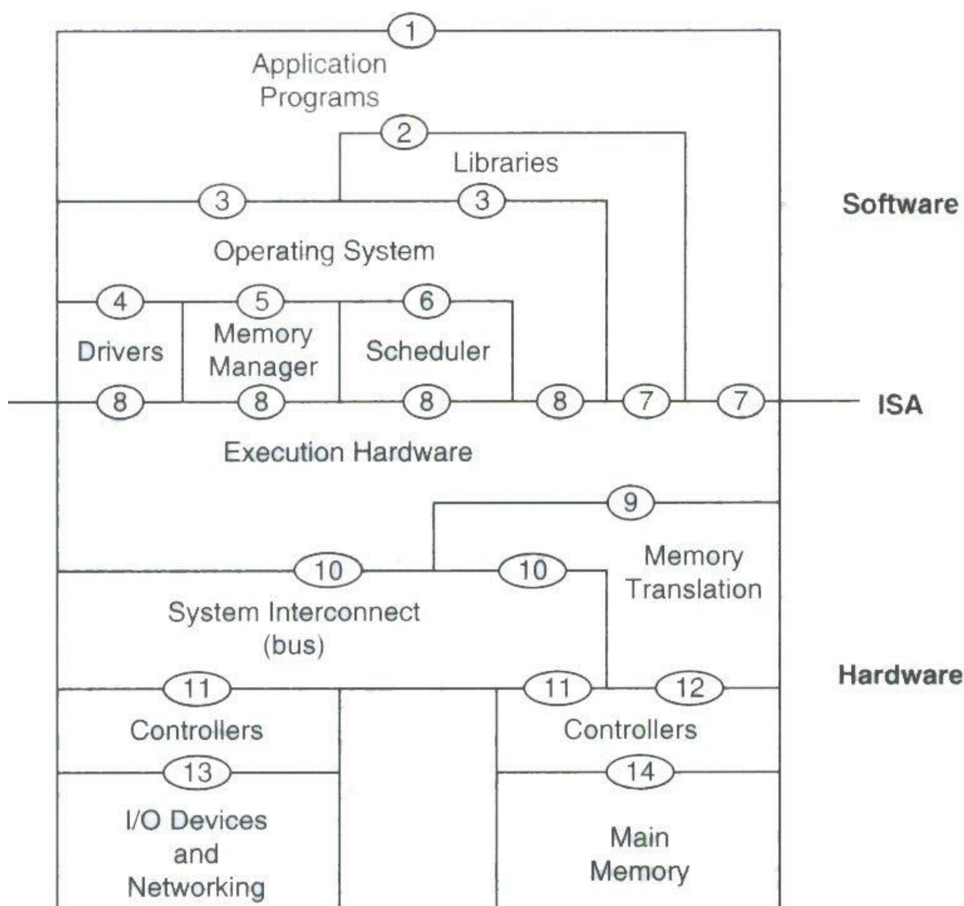
Kapitola třetí je věnována virtuálnímu stroji Parrot. Ve třetí kapitole uvádím, pro které vyšší programovací jazyky je Parrot určený. Interní principy činnosti virt. stroje Parrot a způsob vytváření nových interpretovaných jazyků jsou také popsány.

1 Typy virtuálních strojů

Virtuální stroje můžeme rozlišit podle vrstvy počítačového systému, nad kterou pracují. Vrstva je kontext, ve kterém virtuální stroj běží. Rozlišujeme tyto základní typy virtuálních strojů:

- procesní virtuální stroje *(process virtual machines)*
- systémové virtuální stroje *(system virtual machines)*
- virtuální stroje spoluvyvíjené s hardwarem *(codesigned virtual machines)*
- virtuální stroje zaměřené na multiprocesorovu virtualizaci *(multiprocessor virtualization)*

Na obrázku 1 je znázorněna architektura počítače. Virtuální stroj může pracovat nad instrukční sadou (ISA), což je vrstva označená čísly 7 a 8, a případně nad vyššími vrstvami (zejména nad vrstvou číslo 3 – operačním systémem).



Obrázek 1: Softwarové a hardwarové vrstvy počítače[1]

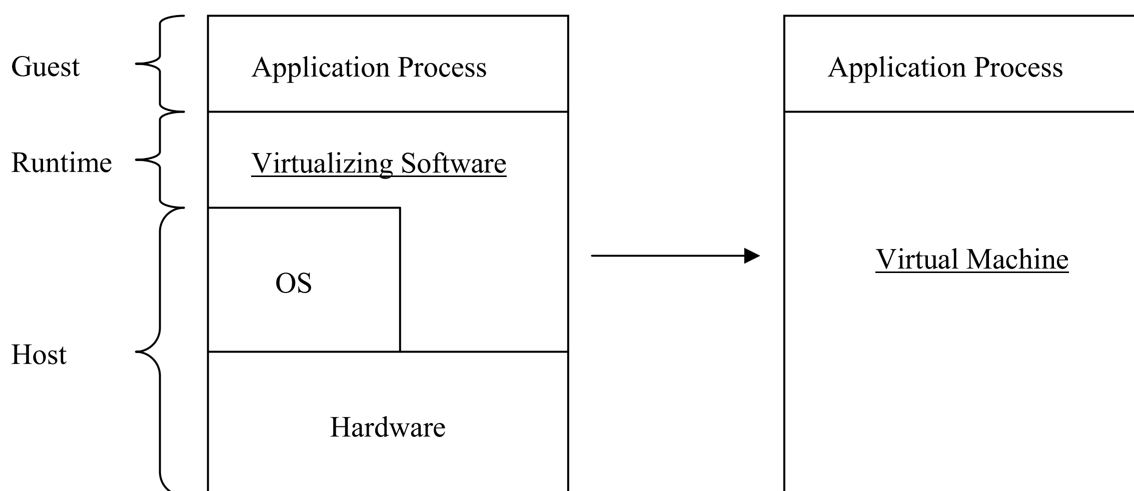
1.1 Procesní virtuální stroj

Procesní virtuální stroj pracuje nad operačním systémem a instrukční sadou. Procesní virtuální stroj slouží k tomu, aby na počítači s danou instrukční sadou, počítačovou architekturou a operačním systémem emuloval program přeložený pro jinou instrukční sadu, architekturu a operační systém.

Od procesního virtuálního stroje se očekává, že hostovanému procesu umožní provedení všech běžných operací v hostitelském systému, což je zejména

- interakce s okolními procesy,
- přístup k datům,
- komunikace se vstupně výstupními zařízeními,
- síťová komunikace.

Obrázek 2 ukazuje virtuální stroj, jak vytváří prostředí hostovanému procesu *(guest, application process)*. Hostovaný proces nerozpozná rozdíl mezi během pod operačním systémem, pro nějž byl přeložen, a během pod virtuálním strojem.



Obrázek 2: Procesní virtuální stroj [1]

1.1.1 Emulace

Proces emulace hostovaného programu prochází několika stádii[1]. Část virtuálního stroje zvaná *loader* na počátku načte binární obraz *(image)* hostovaného programu do vlastní paměti. Loader má dále za úkol inicializaci paměti potřebné pro emulaci. (Kód programu bude později přeložen do cílové instrukční sady a do inicializované paměti se přeložený kód uloží k případnému pozdějšímu použití.) Při inicializaci emulace se také nastavují vektory přerušení *(traps)* operačního systému. Virtuální stroj se primárně snaží využít všech prostředků nabízených hostitelským operačním systémem, což platí i

pro vyvolávání přerušení pro emulovaný proces. (Jestliže dojde k přerušení a emulovaný proces jej očekává, pak k přerušení dojde i u něj.) Emulovaný proces může navíc požadovat i taková přerušení, která mu hostitelský operační systém nenabízí. Návrháři virtuálního stroje se musejí rozhodnout, zda taková přerušení bude virtuální stroj sám simulovat. Takové rozhodnutí souvisí s kompatibilitou prostředí (viz dále).

Po inicializaci přichází na řadu samotná emulace. Emulace zpočátku probíhá tak, že virtuální stroj čte byty zdrojového kódu a rozpoznává výkonné instrukce a data. Po dekódování instrukce dochází k její interpretaci. Virtuální stroj může detekovat blok instrukcí, který bude s velkou pravděpodobností znovu vykonáván. Takovýto blok si virtuální stroj uloží do připravené vyrovnávací paměti *cache*. V případě potřeby si virtuální stroj sáhne do vyrovnávací paměti, místo aby instrukci opět dekódoval. Obvyklé bloky umístěné ve vyrovnávací paměti jsou procedury a části cyklů.

1.1.2 Optimalizace

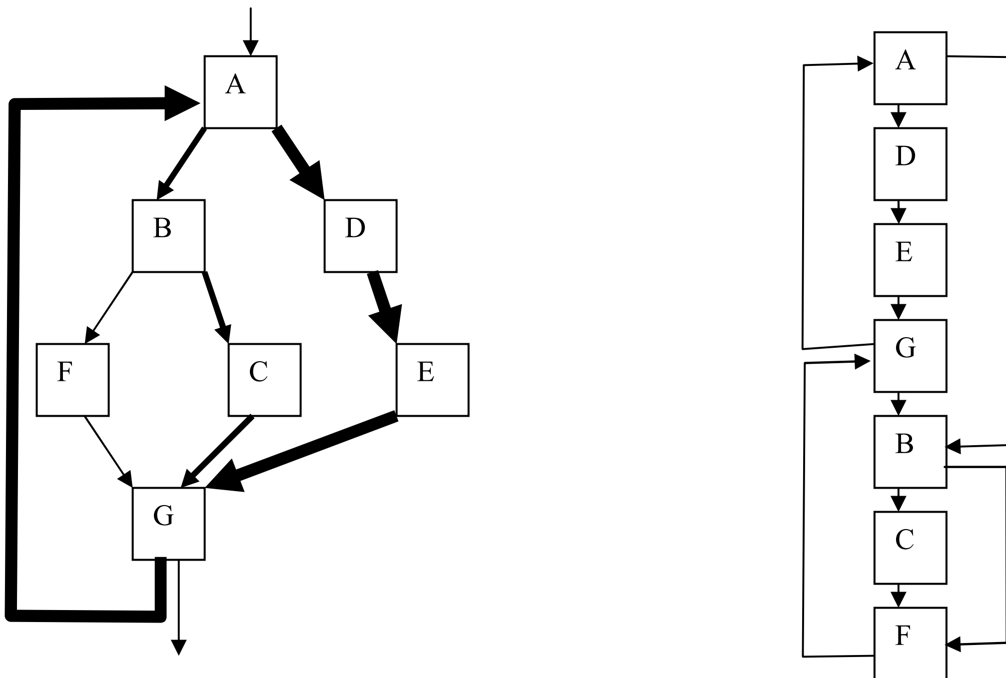
V průběhu emulace dochází k zaplňování vyrovnávací paměti. Vedle samotného emulátoru *(emulation engine)* běží proces nazývaný manažer vyrovnávací paměti *(code cache manager)*. Manažer určuje, které bloky budou ve vyrovnávací paměti dále udržovány, a zajišťuje další jemnější optimalizace.

Bloky instrukcí na sebe často navazují – z jednoho bloku vede skok do druhého bloku nebo druhý blok přímo následuje ve sledu instrukcí po prvním bloku. Bloky si pak lze představit jako uzly orientovaného grafu. Manažer vyrovnávací paměti může měřit, jak často jsou které uzly-bloky navštěvovány. Ze změřených statistik lze vyhodnotit nejběžnější cesty výpočtu.

Nejefektivnější optimalizace je založená na linearizaci nejběžnější výpočetní cesty. Manažer vyrovnávací paměti ukládá bloky na cestě do paměti za sebe. Ve většině počítačů existuje fyzická cache, do které se ukládají nejčastěji navštívené stránky paměti. Pokud jsou bloky na výpočetní cestě v paměti za sebou, pak s velkou pravděpodobností leží ve stejné stránce. Výpočet se tím pádem urychlí, protože nebude docházek k výpadkům stránek.

Uvedená optimalizace patří mezi dynamické optimalizace – prováděné za běhu procesu. Naproti tomu statické optimalizace jsou prováděny při překladu programu. Pojem blok kódu je známý i v souvislosti se statickým překladem. Překladač například vyhodnocuje životnost proměnné uvnitř základního bloku a díky informaci o životnosti rozhoduje o efektivním přidělení registru k proměnné.

Obrázek 3 znázorňuje na levé straně graf bloků s tučně vyznačenou nejběžnější cestou výpočtu. Výhodné uložení bloků v paměti je zobrazeno na pravé straně. Existuje více variant tohoto druhu optimalizace, a proto i uspořádání bloků může být různé (viz [1]).



Obrázek 3: graf bloků kódu (vlevo), výhodné uložení bloků v paměti (vpravo)

1.1.3 Kompatibilita prostředí

V ideálním případě nabízí virtuální stroj hostovanému procesu shodné prostředí (s vyjímkou výkonnosti), pro jaké byl proces přeložen. Ideálního stavu, tedy shody prostředí, nemusí být vždy za každou cenu dosahováno. Emulace hostovaného procesu je časově dosti náročná a dosahování úplné shody v chování vytvářeného prostředí může emulaci neúnosně zpomalit. Návrháři virtuálního stroje se proto předem dohodnou na úrovni kompatibility.

Obecně lze rozdělit virtuální stroje podle kompatibily do dvou kategorií. V první kategorii jsou virtuální stroje s deklarovanou úplnou shodou prostředí. Ve druhé kategorii jsou virtuální stroje s částečnou shodou prostředí. Úplnou shodu prostředí zaručují zejména virtuální stroje spoluvyvíjené s hardwarem *(codesigned virtual machines)*. Pokud se tvůrci virtuálního stroje rozhodnou pro částečnou shodu prostředí, pak obvykle deklarují podmínky, za jakých budou hostované procesy bezchybně běžet. Časté podmínky kladené na hostované programy jsou[1]:

- Program je přeložen určitým konkrétním překladačem.
- Program využívá jen některé knihovny a nebo dodržuje jisté standardy (například POSIX).
- Program vyžaduje omezené systémové prostředky.
- Program dostal osvědčení o bezchybném běhu uvnitř virtuálního systému.

1.2 Systémový virtuální stroj

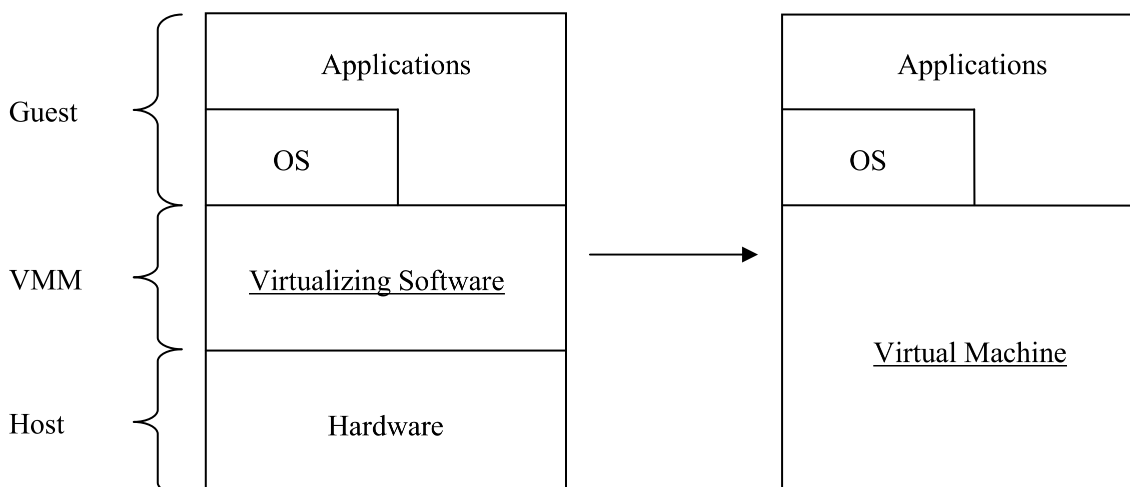
Systémový virtuální stroj pracuje přímo nad hardwarem, resp. přímo nad instrukční sadou. Typickým hostovaným procesem bývá operační systém nebo jiný virtuální stroj nebo i více operačních systémů a virtuálních strojů.

Lze snadno změřit, že hardwarové zdroje na počítači s jednouživatelským operačním systémem, nejsou dostatečně využívány[1]. Aby se hardwarové prostředky využívaly hospodárněji, byly vyvinuty systémy se sdíleným časem *(time sharing systems)*. Systém se sdíleným časem umožňuje souběžnou práci více uživatelům. Každý uživatel má iluzi, že ovládá všechny systémové prostředky.

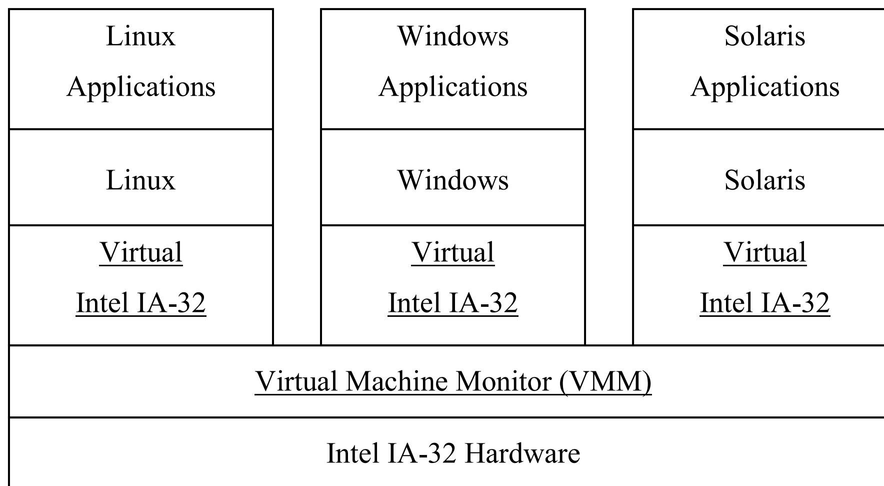
Systémový virtuální stroj vychází z konceptu systému se sdíleným časem. Narozdíl od uživatelů přihlášených do systému to jsou celé operační systémy a virtuální stroje, které pracují nad jedním hardwarem. V systémovém virtuálním stroji se nachází část zodpovědná za rozdělování prostředků mezi hostované systémy. Nazývá se monitor virtuálního stroje *(virtual machine monitor, VMM)*. Monitor volí vhodnou strategii rozdělení prostředků. Některé obvyklé strategie jsou:

- Každý hostovaný operační systém má přidělený svůj hardwarový prostředek.
- Hostované systémy se střídají v přístupu k danému prostředku (časový multiplex).
- Monitor virtuálního stroje sám emuluje zařízení, které není v hardwarové výbavě a hostované systémy jej požadují.

Obrázek 4 zobrazuje virtuální stroj, který vytváří prostředí pro hostovaný operační systém a aplikace v něm běžící.



Obrázek 4: Systémový virtuální stroj[1]



Obrázek 5: Ukázka počítače s architekturou Intel IA-32, který je díky virtuálnímu stroji schopen hostit vnořené operační systémy a jejich aplikace[1].

1.3 Virtuální stroj spoluvyvíjený s hardwarem

Virtuální stroj spoluvyvíjený s hardwarem *(codesigned virtual machine)* se od ostatních zmíněných virtuálních strojů liší tím, že veškerá virtualizace je prováděna hardwarem. Virtualizuje se instrukční sada, tj. zdrojová instrukční sada se pomocí virtualizačního hardwaru převádí na cílovou instrukční sadu.

Historický vývoj instrukčních sad byl z velké části podmíněn ekonomickými tlaky. V dobách, kdy byly hardwarové prostředky nákladné a hardware relativně jednoduchý, dávalo smysl vybavovat instrukční sadu instrukcemi, které reflektovaly používaný hardware. Například pokud byl v počítači akumulátor, pak i v instrukční sadě existovaly instrukce pro manipulaci s akumulátorem. Jak šel vývoj dál, cena hardwaru klesala a zároveň jeho počet v počítači vzrůstal (např. tranzistory na čipu). Nový hardware se podstatně lišil od původního, a proto bylo potřeba modernizovat instrukční sadu, aby lépe odpovídala novým požadavkům. S novou instrukční sadou samozřejmě vyvstal problém – co s aplikacemi přeloženými pro původní instrukční sadu. Tento problém na moderních architekturách řeší specializovaný hardware, který překládá zdrojovou (původní) instrukční sadu do cílové (moderní) instrukční sady. Během vývoje nové instrukční sady se zároveň vyvíjí virtuální stroj, který bude překlad provádět.

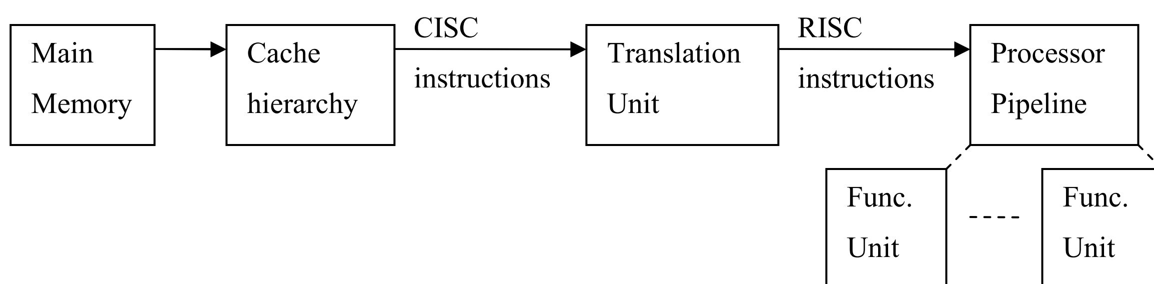
Existují jisté specifické požadavky kladené na překládající virtuální stroj[1]:

- Shoda prostředí musí být úplná (viz kapitola 1.1.3). Všechny instrukce zdrojové sady musejí být správně rozpoznány, hardwarové komponenty odpovídají stejně jako na původním počítači (například výpadky stránek paměti nastanou ve stejném okamžiku).

- Klíčovou vlastností je výkonnost. Zároveň se předpokládá, že virtualizační hardware bude mít nízkou energetickou náročnost.

Podobně jako u procesního virtuálního stroje se používá vyrovnávací paměť *cache* pro přeložený kód. Oproti procesnímu a systémovému virtuálnímu stroji nedochází k virtualizaci žádných jiných hardwarových prostředků kromě procesoru.

Jako příklad spoluvyvíjeného virtuálního stroje uvádím hardware společnosti Intel pro převod instrukční sady CISC do instrukční sady RISC. Tato konverze se provádí na všech moderních Pentiích[2][3]. Proces je naznačen na obrázku 6.



Obrázek 6: Překlad CISC do RISC[1]

1.4 Multiprocesorová virtualizace

Moderní servery, ale i pracovní stanice, dnes obsahují několik procesorů a dostatek paměti. Webové servery spravují velké databáze a přijímají mnoho simultálních požadavků. Vědecká výpočetní centra provádějí náročné výpočty na tisících procesorech. Výpočetní síla postupem času a modernizace roste a více procesorů mají dnes i výkonné osobní počítače. Ne vždy ale aplikace běžící na těchto strojích dokáží využít veškerého výpočetního potenciálu. Počet procesorů často neodpovídá potřebám běžících aplikací. Tento problém lze řešit pomocí multiprocesorové virtualizace – nad polem procesorů je spuštěn manažer virtuálních strojů *(virtual machine monitor, VMM)*. Manažer virtuálních strojů tvoří vrstvu, nad kterou pracují další jednotlivé virtuální stroje. Virtuální stroj nad manažerem může hostovat například webový server. Přicházi-li například webovému serveru mnoho simultálních požadavků, pak manažer webovému serveru dočasně přidělí více procesorů ze svého portfolia. Manažer takto rozkládá zatížení mezi spravované procesory, čímž zvyšuje jejich využití. Manažer obvykle sbírá statistiky o využitelnosti spravovaných prostředků. Statistiky potom slouží jako podklad k rozhodnutí o rozšíření nebo zúžení procesorové základny.

2 Virtuální stroje pro jazyky vyšší úrovně

Virtuální stroje pro vyšší programovací jazyky *(high level language virtual machines, HLL VM)* patří do kategorie procesních virtuálních strojů. Mají za úkol vytvářet prostředí pro aplikace přeložené do kódu (instrukční sady) virtuálního stroje. Mezi známé představitele patří Java VM a Microsoft .NET. Parrot patří také do této kategorie.

Aplikace přeložené do instrukční sady konkrétní počítačové architektury a operačního systému běží pouze v tomto prostředí. Pokud je potřeba tyto aplikace spouštět na jiném systému, pak je vhodné, aby se opět přeložily (portovaly) pro nový systém. Rozhodou-li se programátoři pro portování na novou architekturu, musejí si být vědomi všech odlišných vlastností, například jiných systémových přerušeních, odlišné správy procesů, paměti, periferních zařízeních aj. Alternativně lze aplikaci spustit v procesním virtuálním stroji, jak bylo uvedeno v kapitole 1.1. Avšak i tato alternativa má několik háčků. Procesní virtuální stroj pro nový systém nemusí vůbec existovat nebo nemusí nabízet zcela vyhovující běhové prostředí. Výkon aplikace pod procesním virtuálním strojem samozřejmě poklesne, což také nemusí být žádoucí, jedná-li se o náročné aplikace.

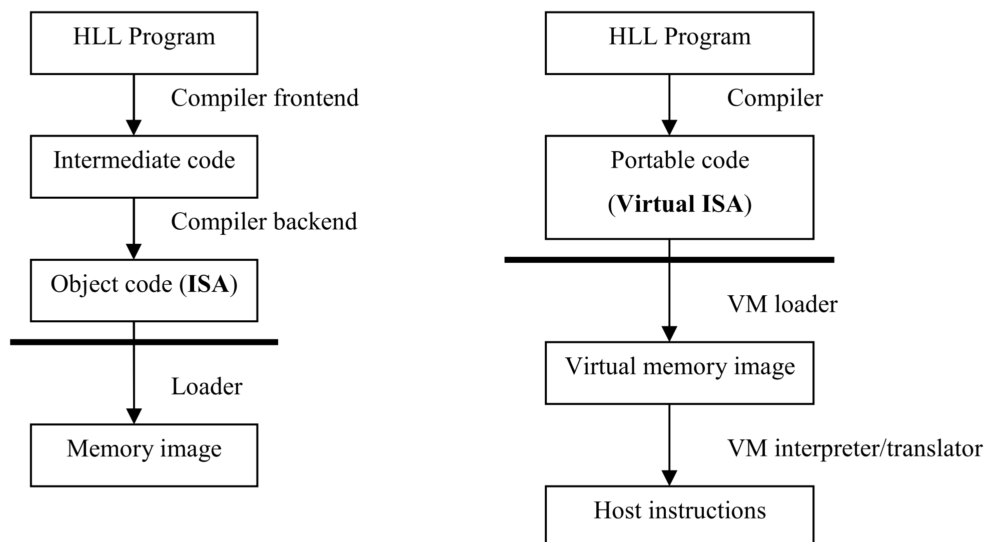
Virtuální stroje pro jazyky vyšší urovně se pokoušejí obejít nutnost portování aplikace pro každou novou instrukční sadu a operační systém. Idea je taková, že aplikace se jednou přeloží a když bude potřeba ji spustit, tak se zavolá HLL VM, aby spuštění provedl. Samotný virtuální stroj musí být samozřejmě portován klasickým způsobem pro každou jednotlivou počítačovou architekturu a operační systém. Prostředí nabízené virtuálním strojem musí být dostatečně robustní. Virtuální stroj vytváří rozhraní nad operačním systémem a zároveň využívá co možná nejvíce funkcí a standardních knihoven hostitelského operačního systému. Na druhou stranu virtuální stroj poskytuje hostované aplikaci jednotný přístup k prostředkům systému. Přístup aplikace k prostředkům systému se děje přes jednotné rozhraní virtuálního stroje. HLL VM také obvykle nabízí bohatou škálu standardních knihovních funkcí. Mnohé z nabízených knihovních funkcí má alternativu v knihovnách hostitelského operačního systému. Ostatní funkce je potřeba explicitně dodefinovat. To, jak hodně virtuální stroj využije hostitelský systém, záleží na konkrétní portaci virtuálního stroje.

2.1 Virtuální instrukční sada

Virtuální stroj (HLL VM) od aplikace požaduje, aby byla přeložena do speciálního přenositelného kódu *(portable code)*. Přenositelný kód je navržený pouze pro interpretaci virtuálním strojem. Nehodí se k přímému spuštění na běžném procesoru a ani k tomu není určen. Existují nicméně pokusy o vytvoření procesoru, který by přijímal přenositelný kód Javy (viz [4], [5], [6], [7]).

Překlad pomocího běžného kompilátoru vegeneruje kód společně s daty. Naproti tomu překlad do přenositelného kódu vygeneruje na jedné straně instrukce, na straně druhé metadata. Specifikace virtuální instrukční sady *(virtual ISA, V-ISA)* a metadat představují podstatnou část specifikace celého virtuálního stroje. Specifikace metadat je často nejdůležitější částí celé dokumentace virtuálního stroje[1]. Vyšší programovací jazyky, překládané do virtuální instrukční sady, bývají nejčastěji objektové (Java, C#). Objekty představují datové struktury vhodné k uložení v metadatech. Metadata pak popisují datové struktury, jejich atributy a vztahy mezi objekty.

Interpretace virtuálních instrukcí se podobá interpretaci instrukcí procesním virtuálním strojem, avšak s tím rozdílem, že interpretace probíhá podstatně rychleji. Podobnost s procesním virtuálním strojem spočívá zejména v optimalizaci přeložených instrukcí. HLL VM si udržuje přeložené instrukce (v kódu hostitelského systému) ve vyrovnávací paměti *(code cache)*. Manažer vyrovnávací paměti může identifikovat bloky kódu, sbírat statistiky a provádět optimalizace vedoucí ke zvýšení výkonnosti aplikace.



Obrázek 7: Srovnání běžného překladu (vlevo) s překladem do přenositelného kódu (vpravo)[1].

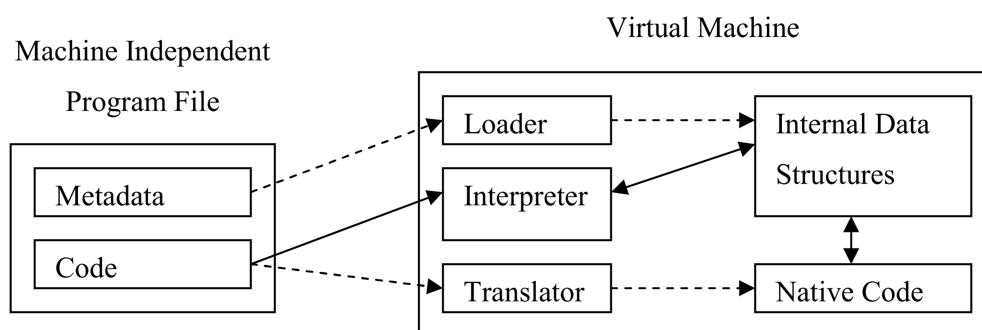
Architektura virtuální instrukční sady je ovlivněna zejména dvěma cíly:

1. virtuální instrukce musejí být přenositelné,
2. interpretace instrukcí musí být co možná nejrychlejší.

Hardware a instrukční sada, nad kterou HLL VM běží mají své specifické rysy, například různý počet registrů. Virtuální instrukční sada musí být dostatečně pružná, aby šla dobře emulovat na jakémkoli hardware. Z toho důvodu jsou všechny dnešní virtuální instrukční sady virtuálních strojů navrženy na bázi zásobníkové architektury. Virtuální instrukce jsou typované – například pro sčítání celých a reálný čísel existují rozdílné kódy.

Doposud jsem opomíjel zdůraznit, že virtuální stroje procesní, systémové a HLL VM sdílejí adresový prostor paměti s emulovaným procesem. Virtuálním strojům činí tato skutečnost poměrně velké potíže. Emulovaný proces musí být virtuálním strojem kontrolován, aby nezasahoval do oblasti paměti obsazené virtuálním strojem. Například běžný program si může dovolit vypočítat adresu, na kterou uloží data. Ošetření podobných akcí může emulaci zpomalit. HLL VM se tomuto problému vyhýbá, protože o umístění dat rozhoduje dynamicky až během interpretace. Ve virtuální instrukční sadě nenajdeme instrukce pro skoky na vypočítanou adresu a instrukce ukazatelové aritmetiky. Kontrolám se ovšem někdy nelze vyhnout – kontrolovány musejí být například meze polí. Virtuální stroj se snaží tyto kontroly co nejvíce zoptimalizovat, tedy omezit. Například před provedením cyklu, který prochází polem, se ověří, že během iterace nedojde k přístupu mimo rozsah pole. Bez optimalizace by se meze pole kontrolovaly při každém přístupu k poli uvnitř cyklu.

Obrázek 8 znázorňuje transformaci přenositelného kódu do kódu hostitelské architektury.



Obrázek 8: Transformace kódu. Čerchované čáry značí transformaci kódu a dat; plné čáry značí přesun kódu a dat během emulace[1].

2.2 Vybrané vlastnosti vituálních strojů pro vyšší programovací jazyky

Ukázal jsem, že klíčové vlastnosti virtuální instrukční sady jsou její přenositelnost a rychlost interpretace virtuálních instrukcí. Interpretované instrukce mohou být dále analyzovány. Jedním důsledkem analýzy instrukcí je optimalizace bloků, o které jsem se již také zmínil. Analýza může navíc sloužit k poznání, že aplikace je šetrná ke svému okolí. V podkapitole 2.2.1 se věnuji tomu, jak virtuální stroj vynucuje ochranu okolí běžící aplikace. V podkapitolách 2.2.2 a 2.2.3 ukážu, že virtuální stroj je schopen dynamicky načítat zdrojové kódy podle potřeby aplikace. Virtuální stroje pro vyšší programovací jazyky od programátora aplikace nevyžadují, aby se staral o správu paměti. Součástí virtuálních strojů bývá garbagge collector.

2.2.1 Bezpečnost

Na klasické počítačové architektuře jsou spouštěné aplikace považovány apriori za rovnocenné. Vyjímku tvoří operační systém a případně softwarové prostředky ovládané operačním systémem, jako například drivery. O tom, co se aplikaci dovolí provádět v systému, rozhoduje převážně operační systému a záleží na operačním systému, jaká pravidla vynucuje. Často je pravidlem, že aplikace dědí práva uživatele, který ji spustil. U některých systémů může být riziko havárie způsobené aplikací větší než u jiných. Aby se předešlo známým katastrofám, mohou v systému paralelně běžet aplikace, které dohlížejí na spuštěné programy. Jsou to například antiviry.

U virtuálních strojů se spouštěné aplikace dělí na důvěryhodné *(trusted)* a nedůvěryhodné *(untrusted)*. Zvláštní ohled je brán na aplikace šířené po internetu. V prostředí Java VM se těmto aplikacím se říká applety. Applety jsou primárně považovány za nejméně důvěryhodné.

Virtuální stroj má přístup k potenciálně všem prostředkům počítače a někdy ani u důvěryhodné aplikace nemusí být žádoucí, aby měla povolen přístup zcela ke všemu. Moderní viruální stroje (Java VM, Microsoft .NET) proto zavádějí míru důvěryhodnosti aplikace. Míru důvěryhodnosti si lze představit tak, že uživatel buď sám určí, co aplikace smí dělat, nebo se spolehne na ověřené informace třetí strany. Uživatel typicky určuje, do jakých adresářů souborového systému bude aplikace moci přistupovat. Ještě jemnějšího povolení přístupu k souborům lze dosáhnout použitím kryptografie. Na kryptografii jsou rovněž založeny certifikáty vydané třetí stranou. Třetí stranou se rozumí uznávaná certifikační autorita (CA). CA ověří záměry aplikace a vydá podepsaný certifikát shody.

Vedle šetrnosti k systémovým prostředkům je virtuálním strojem hlídáno ovlivňování jedné aplikace druhou aplikací. Základní ochrana spočívá v hlídání přístupů aplikace k paměti. Jak jsem uvedl v kapitole o virtuální instrukční sadě, lze tuto ochranu vynutit částečně architekturou instrukční sady. Další kontrolu přístupu do paměti provádí *loader* před interpretací přenositelného kódu (statická kontrola). Nakonec během chodu aplikace jsou prováděny dynamické kontroly.

Statická kontrola má za úkol ověřit, že kód aplikace je konzistentní s datovými strukturami a vazbami mezi strukturami uloženými v metadatech[1]. Kdyby konzistentní nebyly, pak by se instrukce aplikace mohly odkazovat do neznámé paměti a možná ohrožovaly jiné aplikace ve virtuálním stroji nebo virtuální stroj samotný. Mezi statické kontroly také patří ověření skoků. Skoky smějí být pouze relativní vůči čítači instrukcí *(program counter, PC)*. Povolené skokové konstrukce vyšších programovacích jazyků jsou:

- podmíněné skoky (if – then – else)
- výběr z možností (switch)
- volání funkce
- návrat z funkce.

Během dynamické kontroly kódu se ověřují meze polí, přístup k neinicializovaným objektům (tzv. null-referencím) a přetypování na podtřídy[1] *(subclasses)*.

Prostředí, kde je aplikaci povoleno operovat se nazývá *sandbox*. Programu je povoleno provádět libovolné akce uvnitř sandboxu. K prostředkům mimo sandbox se aplikace dostane pouze s explicitním povolením. V prostředí Microsoft .NET se kódu uvnitř sandboxu říká *managed code*[8]. Naopak kód mimo sandbox se nazýva *unmanaged code*.

K dodržování bezpečnosti aplikace významně přispívá to, že aplikace jsou napsané v objektovém silně typovaném vyšším programovacím jazyce (jako např. Java nebo C#). Program v takové jazyce musí jasně deklarovat jmenný prostor požadovaného prostředku (objektu).

2.2.2 Robustnost

Předpokládá se, že v prostředí virtuálního stroje poběží i rozsáhlé aplikace. Praxe ukazuje, že produktivitu vývoje náročných aplikací lze zvýšit kombinováním vlastního kódu se znovupoužitelným kódem. Znovupoužitelný kód se dá distribuovat pomocí objektů, modulů, knihoven, balíčků, komponent apod. Výrobci virtuálních strojů jdou této potřebě vstříc – vyšší programovací jazyky, ze kterých jsou aplikace pro virtuální stroje přeloženy, jsou založeny na

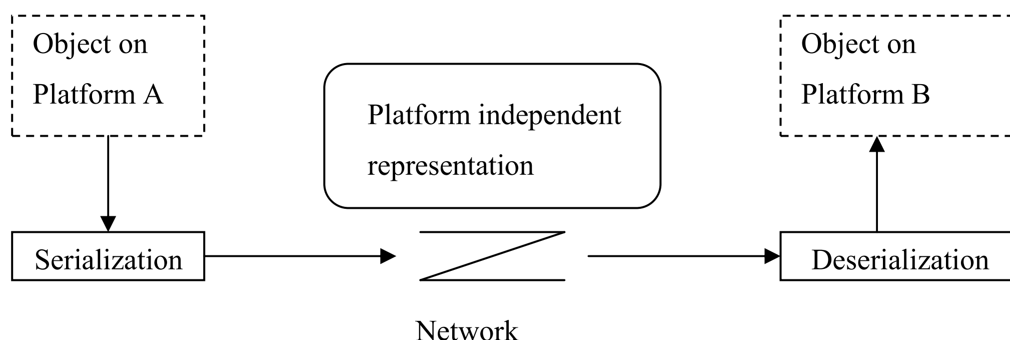
objektově orientovaném paradigmatu. Další vlastnosti, které přispívají k robustnosti, jsou silné typování a automatická správa paměti[1].

2.2.3 Síťování

Běh programu může záviset na parametrech, s jakými byl spuštěn, nebo na vstupech z okolí. Mějme například aplikaci, která může s uživatelem komunikovat jednak přes textový terminál a jednak přes vlastní grafické uživatelské rozhraní. Aplikace je přeložená do přenositelného kódu virtuálního stroje (např. Java VM nebo Microsoft .NET) a uživatel aplikaci spustí v textovém režimu. Za běhu aplikace nebude nikdy využita grafická knihovna a virtuální stroj ji tedy ani nenačte do paměti. Takto se ušetří paměť pro jiné procesy. Moderní virtuální stroje nepožadují, aby byla celá aplikace načtena v paměti. Místo toho se moduly aplikace načítají inkrementálně dle potřeby. Tento způsob spouštění programů nalézá uplatnění u aplikací spouštěných ze sítě. Po síti cestují jen aktuálně použité moduly. Šetří se tak síťový provoz.

Mají-li dvě aplikace komunikovat po síti, musejí se dohodnout na struktuře přenášených dat. Objektové prostředí vybízí k výměně dat v podobě objektů. Virtuální stroje umožňují takovou výměnu a interně se starají o to, aby přenášená data měla jednotný formát na všech počítačových architekturách. Odeslaný objekt je vysílajícím virtuálním strojem transformován do jednotného formátu. Přijímající virtuální stroj přijme data v jednotném formátu, transformuje je do objektů a předá přijímající aplikaci. Procesu transformace dat se říká serializace resp. deserializace. Serializace slouží i k uložení dat na perzistentní médium, například pevný disk. Obrázek 9 znázorňuje proces serializace.

Aplikace, která přijme objekt, má pravděpodobně v úmyslu využít i funkce definované pro tento objekt. Po síti jí nicméně zatím přišla pouze data. Přijímající aplikace tedy požádá o dodání kódu přidruženého k přijatému objektu. Následně je aplikaci dodán i kód. Přijímající virtuální stroj zanalyzuje kód (viz kapitola 2.2.1) a povolí vykonání funkcí objektu. Načtení a analýza kódu (funkcí i dat) za běhu se nazývá reflexe.



Obrázek 9: Serializace a deserializace objektu[1]

3 Virtuální stroj Parrot

Virtuální stroj Parrot patří mezi virtuální stroje pro vyšší programovací jazyky (HLL VM). Virtuální stroje z této rodiny, jako Java VM nebo Microsoft .NET, podporují staticky typované programovací jazyky. Parrot je naproti tomu navržen pro efektivní běh aplikací napsaných v dynamických jazycích. Mezi dynamické jazyky patří například Perl a Python. U dynamicky typovaných jazyků lze měnit typový systém za běhu aplikace, rozšiřovat aplikaci o nový kód, definovat funkce vyšších řádů.

Parrot přijímá zdrojový kód v několika formátech. Součástí virt. stroje Parrot je překladač. Překladač transformuje zdrojový kód do přenositelného kódu *(bytecode)*. Přenositelný kód se skládá z virtuálních instrukcí virt. stroje Parrot. Na rozdíl od virtuálních strojů podporujících staticky typované programovací jazyky je architektura virtuální instrukční sady registrová (srovnejte se zásobníkovou u Javy nebo Microsoft .NET). Vlastnostmi instrukční sady Parrotu a jejich interpretací se zabývá následující kapitola.

3.1 Interpret

Interpret přenositelného kódu je navržen tak, aby se podobal procesoru počítače. U virtuálníh strojů jako Java VM nebo Microsoft .NET je virtuální instrukční sada založena na zásobníkové architektuře. Instrukční sada Parrotu je naopak architektura registrová. Idea je taková, že přiblížení přenositelného kódu skutečnému hardwaru, nad kterým Parrot běží, povede k efektivnějšímu překladu do strojového kódu počítače. Přiblížení architektury instrukční sady musí být do jisté míry omezené, aby nenastávaly problémy s přenositelností kódu. Přenositelnost je stále klíčovou vlastností.

Práce s registry je ve virt. stroji Parrot rozšířena oproti práci s běžnými hardwarovými registry. Parrot rozlišuje čtyři druhy registrů: celočíselné registry, registry pro čísla s plovoucí řádovou čárkou, řetězcové registry a nakonec objektové registry. Další rozdíl oproti hardwaru spočívá v tom, že Parrot umožňuje aplikaci přistupovat k neomezeně mnoha registrům každého typu. Zvyšuje se tím komfort při psaní zdrojového kódu uživatelem. Registry jsou uloženy v registrových rámcích. Rámce se nacházejí na rámcovém zásobníku. Každá spuštěná funkce nebo navštívený blok (během interpretace) má na zásobníku vlastní registrový rámec. Parrot s rámci manipuluje automaticky bez zásahu uživatele.

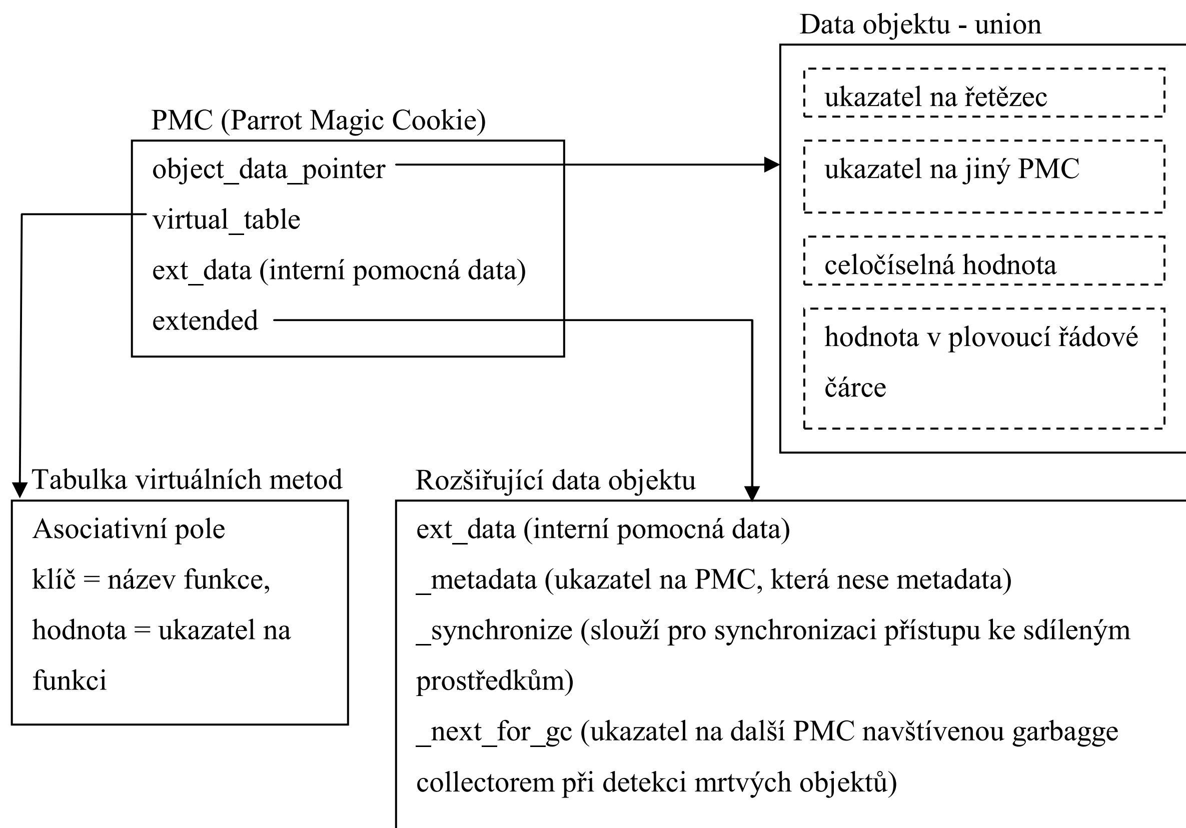
Instrukce virt. stroje Parrot jsou typované (podobně jako instrukce ostatních virtuálních strojů) a uzpůsobené pro práci s registry. Instrukce pro manipulaci s daty přijímají vždy alespoň jeden registr jako svůj parametr. Zvláštností instrukcí virt. stroje Parrot je to, že všechny vracejí adresu následující

instrukce v bytecodu. Když chce interpret skočit na následující instrukci, nemusí vyhledávat délku aktuální instrukce (včetně argumentů) v tabulce instrukcí. Interpretace se tím urychluje.

3.2 Reprezentace objektů

Zvláštní pozornost mezi registry Parrotu si zaslouží objektové registry. V terminologii Parrotu se nazývají PMC *(Parrot Magic Cookie)*. Interní reprezentace objektu obsahuje struktury, které umožňují implementaci objektových vlastností – dědičnosti, polymorfismu, zapouzdření. Při vytvoření nového objektu se inicializuje tabulka virtuálních metod. Do tabulky se implicitně přidají ukazatele na interní funkce, které zajistí objektové chování PMC. Parrot navíc nabízí funkce pro manipulaci s interní reprezentací objektů. Lze jimi měnit chování objektů za běhu. Tímto způsobem je v Parrotu dosaženo dynamické změny typového systému.

Obrázek 10 ukazuje interní reprezentaci objektu v Parrotu. Podobně jako u jiných virtuálních strojů, tak i u Parrotu jsou součástí objektu metadata (uložená v rozšiřujících datech objektu).



Obrázek 10: Interní reprezentace objektu ve virt. stroji Parrot

3.3 Překladač

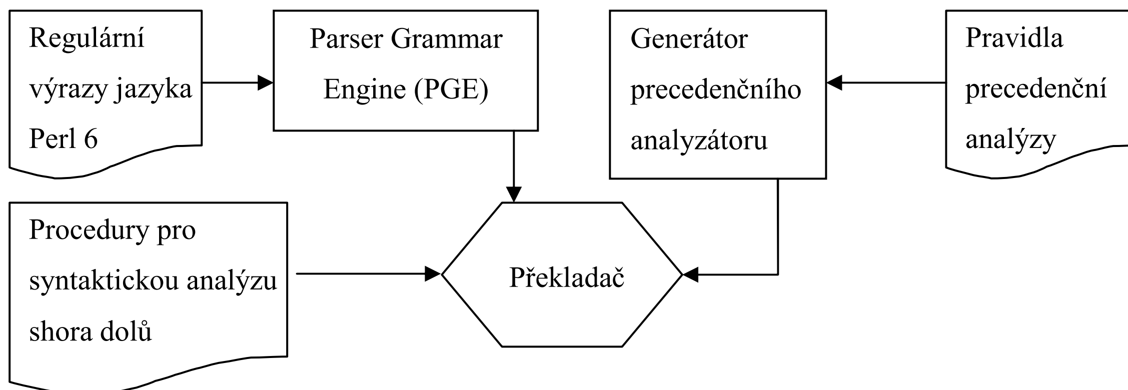
Parrot obsahuje překladač vyššího a nižšího programovacího jazyka. Alternativně lze Parrotu předat abstraktní syntaktický strom, který vznikl analýzou cizím překladačem. U všech tří podob vstupu se předpokládá, že budou generovány z jiného programovacího jazyka. Parrot navíc nabízí generátor kompilátorů, podobný generátorům yacc nebo bison, který by měl usnadnit vývoj kompilátoru z cizího programovacího jazyka. Jazyky rozpoznávané překladačem virt. stroje Parrot jsou:

- PIR (Parrot intermediate representation). PIR je vyšší programovací jazyk. Programy v PIR může psát i člověk, což je výhodné při ladění chyb.
- PASM (Parrot Assembler). PASM je nižší programovací jazyk. Programy v PASM mohou být člověku srozumitelné, není ale účelem, aby je člověk v PASM psal.
- PAST (Parrot Abstract Syntax Tree). Máme-li k dispozici již hotový překladač vlastního jazyku, pak si můžeme ušetřit práci s generováním cílového kódu. Po ukončení syntaktické analýzy předáme virt. stroji Parrot derivační strom. Designéři virt. stroje Parrot chtějí, aby Parrot následně sám sestavil cílový kód, zatím je tato iniciativa nereailizována. Parrot v současnosti pouze interpretuje získaný strom.

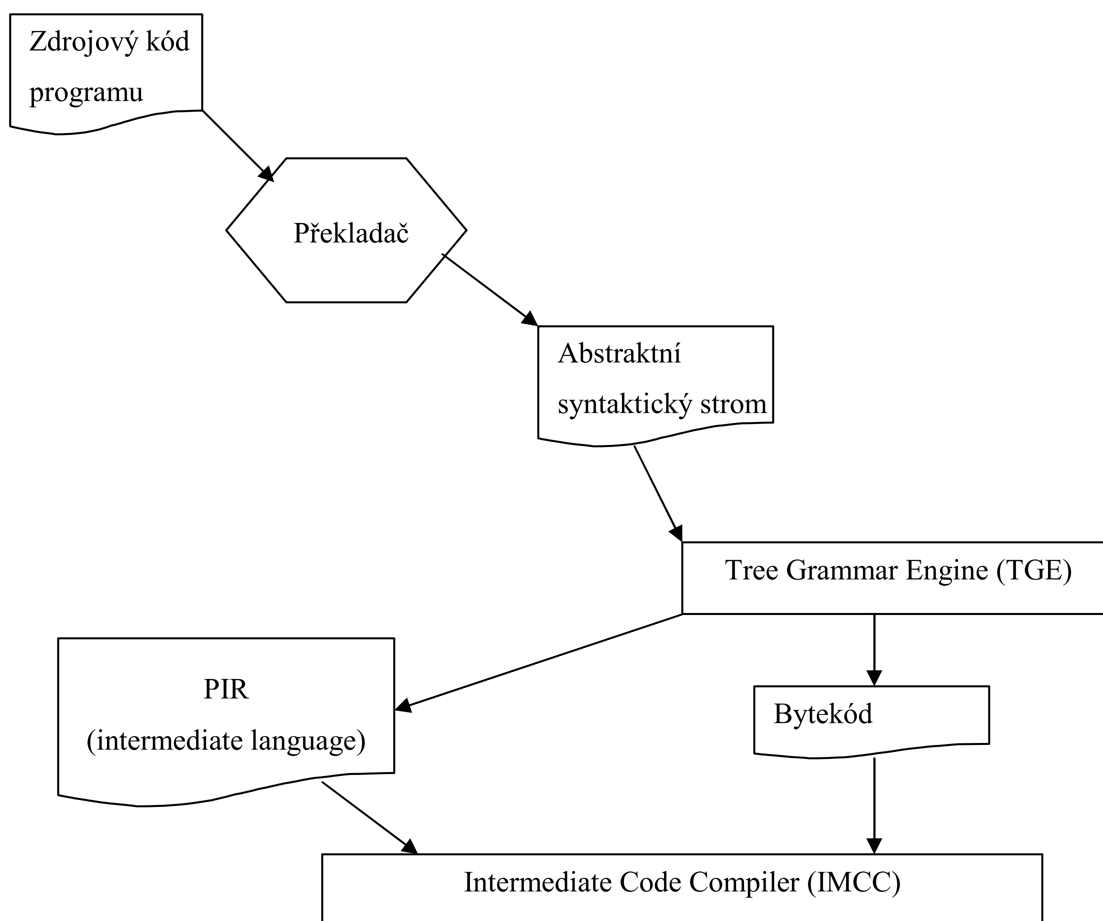
Parrot nabízí nástroje pro snadný návrh vlastního překladače. Prvním nástrojem je Parser Grammer Engine (PGE). PGE přijímá rozšířené regulární výrazy ve formátu Perl 6. (Perl 6 vytváří abstrakci nad regulárními výrazy: rozlišuje běžné regulární výrazy, tokeny a pravidla. Rozšířené regulární výrazy jazyku Perl 6 jsou výpočetně ekvivalentní konečným automatům.) PGE transformuje regulární výrazy do procedur jazyka PIR. Soubor vygenerovaných procedur je schopen provádět lexikální analýzu zdrojového textu. (Analogie s programem Lex je zřejmá.) Po člověku navrhujícím překladač se následně žádá, aby do zdrojového kódu dopsal nové procedury, které budou za pomoci původních procedur provádět syntaktickou analýzu shora dolů. Syntaktickou analýzu shora dolů lze doplnit o precedenční analýzu (např. matematických výrazů) zdola nahoru. Parrot obsahuje nástroj pro vygenerování precedenčního analyzátoru. Programátor opět obdrží procedury, které začlení do zdrojového kódu. Výsledný překladač přijímá zdrojový text a vrací derivační strom.

Derivační strom jde na vstup nástroji Tree Grammar Engine (TGE). TGE je nástroj pro optimalizaci získaného derivačního stromu. TGE vrátí kód v PIR nebo bytekód virtuálního stroje (záleží na nastavení). PIR je překládán nástrojem Intermediate Code Compiler do bytekódu a následně interpretován.

Na obrázcích 11 a 12 je znázorněna tvorba překladače a překlad do kódu virt. stroje Parrot.



Obrázek 11: Tvorba překladače s pomocí prostředků virt. stroje Parrot.



Obrázek 12: Překlad a interpretace programu.

Intermediate Code Compiler (IMCC) v sobě obsahuje několik vnitřních částí:

- Parser PASM a PIR – přijímá zdrojový kód a emituje derivační strom.
- Kompilátor bytekódu – přijímá derivační strom od parseru a emituje neoptimalizovaný proud instrukcí. Instrukce lze okamžitě interpretovat, nicméně je lepší spustit optimalizátor k provedení statické optimalizace.
- Optimalizátor – přijímá proud instrukcí a optimalizuje je.
- Interpret – přijímá proud instrukcí od optimalizátoru nebo kompilátoru bytekódu a provádí jejich interpretaci

Na jednotlivé části IMCC lze pohlížet jako na moduly, které lze dle potřeby vyměnit, nebo každý modul nechat běžet odděleně – samostatně.

Parrot nabízí užitečná rozšíření (moduly) nad rámec virtuálního stroje *(Extensions)*. Mezi standardní moduly patří:

- Vstupně-výstupní podsystém *(I/O subsystem)*. Každá instalace Parrotu pro různé platformy má vlastní vstupně-výstupní podsystém pro systémově závislou komunikaci s periferiemi mimo prostředí Parrotu. Součástí Parrotu není modul pro serializaci do jednotného formátu dat.
- Modul pro zpracování regulárních výrazů *(Regular expression engine)*. Zpracování regulárních výrazů je hojně využíváno v nástroji Parser Grammer Engine při analýze zdrojového kódu. Dynamické jazyky jako Perl nebo Python jsou historicky zaměřené na analýzu textů, kde regulární výrazy hrají také významnou roli. Z těchto důvodů je nástroj pro zpracování regulárních výrazů veden zvlášť jako přídavný modul.
- Modul transformace dat *(Data transformation engine)*. Jádro modulu je shodné s jádrem nástroje Tree Grammar Engine. Mezi datové transformace patří například XSLT transformace.

4 Závěr

V této práci jsem se seznámil s nejvýznamnějšími virtuálními stroji současnosti. O každém virtuálním stroji jsem pojednal tak, abych čtenáři poskytl základní, avšak ucelený přehled.

Významná část mé práce je věnována virtuálním strojům pro vyšší programovací jazyky. Odvětví těchto virtuálních strojů je stále oblíbenější jak mezi programátory, kterým nabízí prostředí moderních vyšších programovacích jazyků, tak mezi výzkumnými pracovníky, pro které může být výzvou například optimalizace interpretů.

Jako pozitivní spatřuji snahu producentů virtuálních strojů pro vyšší programovací jazyky v otevření architektur virtuálních strojů širší odborné veřejnosti, nejenom vědcům. Důkazem této snahy je například Parrot, do kterého lze doimplementovat překladač vlastního jazyka. Microsoft rovněž dovoluje překládat vlastní jazyky do bytekódu virtuálního stroje Microsoft .Net (viz [10]).

Kapitola věnovaná virtuálnímu stroji Parrot pojednává o vniřních principech virtuálního stroje pro dynamicky typované jazyky. Přínosná může být zejména proto, že na trhu je nedostatek literatury o této problematice. (Většina knih se zabývá staticky typovanými jazyky a jejich virtuálními stroji). Důkladnější zdokumentování nástrojů a interních záležitostí Parrotu by mohlo být námětem na další práci.

V praktické části bakalářské práce se věnuji výstavbě překladače jednoduchého objektového jazyka do kódu Parrotu. Registrová architektura Parrotu je intuitivní k pochopení a významně mi usnadnila práci.

Příloha 1: Návrh jazyka

Navrhnul jsem a implementoval překladač vlastního jazyka. Jazyk je objektový, strukturou se podobá vyšším moderním programovacím jazykům jako např. C++. Jazyk je silně typovaný. Program se skládá z definic funkcí a tříd, z deklarací proměnných, z příkazů (volání funkcí), výrazů, podmínek a cyklů. Deklarace proměnných, příkazy a výrazy jsou ukončeny středníkem.

Proměnné musejí být nejprve deklarovány, než budou použity. Základní datové typy jsou int (celá čísla), double (desetinná čísla), bool (pravda/nepravda) a string (řetězce). Proměnné mohou být objekty – instancemi tříd. Mezi složené datové typy patří také pole. Deklarace proměnné se smí objevit uvnitř funkce nebo třídy. Globální proměnné nejsou povoleny. Deklarace proměnné může zastínit stejně pojmenovanou proměnnou definovanou v nadřazeném kontextu. V jednom kontextu (ve funkci nebo v třídě) je povolena nejvýše jedna deklarace proměnné daného jména. K proměnných se přistupuje tak, že se před proměnnou napíše dolar ($). V třídě je implicitně deklarována proměnná *this*, která ukazuje na objekt. K *this* se přistupuje bez předřazeného dolaru. Proměnné základních datových typů (int, double, bool, string) jsou předávány hodnotou. Proměnné jiných typů jsou předávány odkazem. Pojem ukazatel v tomto jazyku neexistuje.

Příklady deklarací proměnných (T je třída): int i; int j; string s; T obj;

Příklad použití proměnných: $i = $j + 1;

Funkce jsou definovány buď jako globální nebo jako člensk é ve třídě. Funkce musí mít specifikován návratový typ nebo void. Funkce smí přijímat 0 a více parametrů oddělených čárkami. Tělo funkce nesmí být prázdné. Návrat hodnoty se provádí příkazem return výraz;. Funkce mohou být přetížené. Při určování správné funkce záleží na typech parametrů.

Příklad funkce: int foo(int i, int j) { int x; $x = $i + $j; return $x; }

Třídy jsou definovány jako globální. Vnořené definice tříd nejsou přípustné. Třídy mohou dědit od jiných tříd. Děděné třídy se uvádějí za název a navzájem jsou odděleny čárkou. Proměnné a funkce definované ve třídě jsou všechny veřejné a virtuální (viz C++). Tělo třídy nesmí být prázdné.

Příklad: class Animal { int legs; void SetLegs(int i) { $legs = $i;} }

class Cow Animal { void Say() { print(“Cow has ”); print(legs); print(“ legs.\n”); } }

Podmínky mají tvar: „if (*podmínka*) { … } else { … }” nebo bez části s else.

Cykly jsou s podmínkou na začátku: “while(*podmínka*) {…}” nebo s podmínkou na konci: „do {…} while(*podmínka*)“. Z cyklu lze vyskočit příkazy break a continue.

Předdefinované funkce jsou: int readint(), double readdouble(), string readstring() pro čtení ze standardního vstupu, void print(int i), void print(double d), void print(string s) pro výstup. Rozpoznávané konstanty jsou: celá čísla, čísla s desetinnou tečkou a řetězce uzavřené do uvozovek.

Příloha 2: Gramatika jazyka

```
program: statement_seq        Program je sekvence výrazů.
      ;

statement_seq: statement_seq statement {
      | statement
      ;
```

Výrazy jsou:

```
statement: expression_statement  vyhodnocované výrazy a příkazy
      | selection_statement   podmínky
      | iteration_statement   cykly
      | jump_statement        skoky (break, continue, return)
      | declaration_statement deklarace
      | compound_statement    složené výrazy.
      ;

compound_statement: '{' statement_seq '}'
      ;
expression_statement: expression ';'  Přikazy jsou ukončené středníkem.
      ;
```

Podmínky

```
%nonassoc LOWER_THAN_ELSE;
%nonassoc ELSE;
selection_statement: IF '(' condition ')' statement %prec LOWER_THAN_ELSE
      | IF '(' condition ')' statement ELSE statement
      ;
```

Výraz vyhodnocovaný v podmínce smí být cokoliv kromě přiřazení.

```
condition: constant_expression
      ;
```

Cykly s podmínkou na začátku a na konci.

```
iteration_statement: WHILE '(' condition ')' statement
      | DO statement WHILE '(' condition ')' ';'
      ;
```

Skoky

```
jump_statement: BREAK ';'
      | CONTINUE ';'
      | RETURN ';'
      | RETURN constant_expression ';'
      ;
```

Deklarace

```
declaration_statement: variable_declaration
      | function_declaration
      | class_declaration
      ;
```

Typy proměnných jsou:

```
type_specifier: ID      třídy,
      | INT             celá čísla,
      | STRING          řetězce,
      | DOUBLE          čísla s desetinnou tečkou,
      | BOOL            pravda / nepravda,
      | VOID            prázdný typ,
```

```
        | type_specifier '[' constant_expression ']'
```
pole s daným počtem prvků,
```
        | type_specifier '[' ']'
```
pole.
```
        ;
```

Deklarace proměnné
```
variable_declaration: type_specifier ID ';'
        ;
```
Parametry funkce v definici funkce
```
params: params ',' type_specifier ID
        | type_specifier ID
        |
        ;
```
Definice funkce
```
function_declaration: type_specifier ID '(' params ')' '{' statement_seq
'}'
        ;
```
Definice třídy
```
class_declaration: CLASS ID  base_list '{' statement_seq '}'
          ;
```

Seznam tříd, od kterých se dědí
```
base_list: ID
        | base_list ',' ID
        |
        ;
```

Literály
```
literal: INT_VAL
        | STRING_VAL
        | DOUBLE_VAL
        | BOOL_VAL
        | NULL_VAL
        ;
```
Základní výrazy je:
```
primary_expression: literal        literál,
        | THIS                     this,
        | '(' expression ')'       uzávorkovaný výraz
        | '$' ID                   proměnná.
        ;
```

Postfixový výraz je:
```
postfix_expression: primary_expression    základní výraz
        | postfix_expression '[' constant_expression ']' přístup k prvku pole
        | ID '(' ')'                       volání funkce bez parametrů
        | ID '(' param_list ')'            volání funkce s parametry
        | postfix_expression '.' '$' ID    přístup k členské proměnné
        | postfix_expression '.' ID '(' param_list ')' volání členské funkce
        | postfix_expression '.' ID '(' ')' volání členské funkce bez parametrů
        ;
```

Seznam parametrů volání funkce
```
param_list:  constant_expression
        | param_list ',' constant_expression
        ;
```

Unární výraz
```
unary_expression: postfix_expression
        | unary_operator cast_expression
        ;
```

Unární operátory jsou plus a minus.

```
unary_operator: '+'
      | '-'
      ;
```

Přetypování

```
cast_expression: unary_expression
      | '(' type_specifier ')' unary_expression
      ;
```

Výraz s prioritou násobení

```
multiplicative_expression: cast_expression
      | multiplicative_expression '*' cast_expression
      | multiplicative_expression '/' cast_expression
      ;
```

Výraz s prioritou sčítání

```
additive_expression: multiplicative_expression
      | additive_expression '+' multiplicative_expression
      | additive_expression '-' multiplicative_expression
      ;
```

Relační výraz s prioritou menší než

```
relational_expression: additive_expression
      | relational_expression '<' additive_expression
      | relational_expression '>' additive_expression
      | relational_expression LE additive_expression
      | relational_expression GE additive_expression
      ;
```

Relační výraz s prioritou rovnosti

```
equality_expression: relational_expression
      | equality_expression EQ relational_expression
      | equality_expression NE relational_expression
      ;
```

Logická konjunkce

```
logical_and_expression: equality_expression
      | logical_and_expression AND equality_expression
      ;
```

Logická disjunkce

```
logical_or_expression: logical_and_expression
      | logical_or_expression OR logical_and_expression
      ;
```

Přiřazení

```
assignment_expression:  logical_or_expression
      | logical_or_expression '=' assignment_expression
      ;
```

Obecný výraz

```
expression: assignment_expression
      | expression ',' assignment_expression
      ;
```

Konstantní výraz – cokoliv kromě přiřazení

```
constant_expression: logical_or_expression
      ;
```

Literatura

[1] Smith, J. E., Nair, R. Virtual Machines: Versatile Platforms for Systems and Processes, San Francisco, Elsevier, 2005.

[2] Hinton, G. et al. The Microarchitecture of the Pentium 4 Processor, *Intel Technology Journal,* 2005.

[3] Keltcher, C. N. et al. The AMD Opteron Processor for Multiprocessor Servers, *IEEE Micro* (March / April) 2005, p. 66-70.

[4] Processor Technology Resources – picoJava, 2008, http://www.sun.com/software/communitysource/processors/picojava.xml

[5] Drábek, V.**:** A Model of PicoJava CPU Core, *MOSIS'99*, Vol. 2, Rožnov p. Radhoštěm, CZ, MARQ, 1999, p. 27-29

[6] JOP – Java Optimized Processor, http://www.jopdesign.com/, 2008

[7] McGhan, H, O'Connor, M PicoJava: A Direct Execution Engine for Java Bytecode, *IEEE Computer* (October), 1999

[8] Standard ECMA-335: Common Language Infrastructure (CLI) 4th edition, 2006, http://www.ecma-international.org/publications/standards/Ecma-335.htm

[9] Parrot Documentation, www.parrotcode.org, 2008

[10] Nigles, G. E.: Build Your Own .NET Language and Compiler, Apress, 2004

[11] Aho, A. V., et al.: Compilers: Principles, Techniques & Tools, Addison Wesley, 1988

Seznam příloh

Příloha 1. Návrh jazyka

Příloha 2. Komentovaná gramatika jazyka

Příloha 3. CD s digitální podobou bakalářské práce a zdrojovými texty překladače