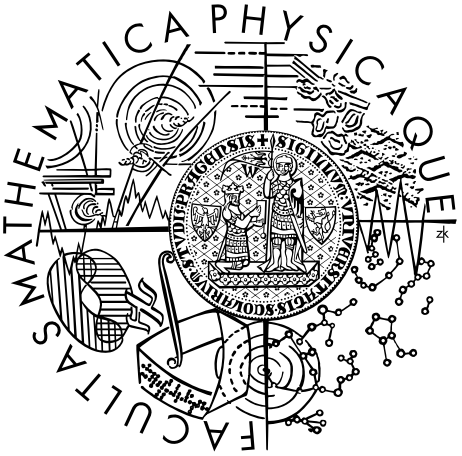
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Petr Stefan

Zhodnocení použitelnosti Raspberry Pi pro distribuované výpočty

Katedra softwarového inženýrství

Vedoucí bakalářské práce: RNDr. Martin Kruliš, Ph.D.

Studijní program: Informatika

Studijní obor: Obecná informatika

Praha 2015

Prohlašuji, že jsem tuto bakalářskou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V Praze dne 29. července 2015 Petr Stefan

Název práce: Zhodnocení použitelnosti Raspberry Pi pro distribuované výpočty

Autor: Petr Stefan

Katedra: Katedra softwarového inženýrství

Vedoucí bakalářské práce: RNDr. Martin Kruliš, Ph.D., Katedra softwarového inženýrství

Abstrakt: Cílem práce bylo zhodnotit výkonnostní vlastnosti mikropočítače Raspberry Pi, a to především z hlediska možnosti sestavení výpočetního klastru z více těchto jednotek. Součástí práce bylo vytvoření sady testů pro měření výpočetního výkonu procesoru, propustnosti operační paměti, rychlosti zápisu a čtení z persistentního úložiště (paměťové karty) a propustnosti síťového rozhraní Ethernet. Na základě porovnání výsledků získaných měřením na Raspberry Pi a na vhodném reprezentativním vzorku dalších běžně dostupných počítačů bylo provedeno doporučení, zda je konstrukce Raspberry Pi klastru opodstatněná. Tyto výsledky rovněž poskytly přibližné ekonomické a výkonnostní projekce takového řešení.

Klíčová slova: Raspberry Pi výkon distribuovaný výpočet benchmark

Title: Assessing Usability of Raspberry Pi for Distributed Computing

Author: Petr Stefan

Department: Department of Software Engineering

Supervisor: RNDr. Martin Kruliš, Ph.D., Department of Software Engineering

Abstract: The main objective of the presented work is to assess performance of Raspberry Pi microcomputer and its applicability as a work node of a computational cluster. The thesis contains implementation of set of tests that evaluate the CPU performance as well as the throughput of operation memory, persistent storage (SD card), and Ethernet network adapter. The results of comparison of empirical results obtained from Raspberry Pi and representatives of regular computers provided basis for recommendation, whether a construction of Raspberry Pi cluster is feasible. Furthermore, the results allowed us to make an estimation of economical and performance aspects of the solution.

Keywords: Raspberry Pi performance distributed computing benchmark

Děkuji RNDr. Martinu Krulišovi, Ph.D. za pomoc, podnětné připomínky a pečlivé vedení této práce. Rád bych také na tomto místě poděkoval svému otci, RNDr. Oto Stefanovi, za jeho rady při studiu i tvorbě tohoto textu a slečně Bc. Markétě Holé za pomoc při tvorbě obrázků.

Obsah

Úvod

Důležitým směrem ve vývoji počítačové techniky je paralelismus. Více výpočetních jader umožňuje spouštění několika samostatných programů současně nebo zrychlení výpočtu jediného programu. Jednou z možností, jak dosáhnout paralelního zpracování, je vytvoření distribuovaného výpočetního zařízení. Jedná se o kolekci nezávislých počítačů, které spolupracují na řešení zadaných úloh. Celý systém je pro vnější pozorovatele koherentní, jeho vnitřní struktura může zůstat uživatelům skryta. Jednotlivé uzly jsou propojeny některou z komunikačních technologií a zpravidla si vyměňují data pomocí zpráv. Takováto zařízení běžně nazýváme počítačovými clustery. Sofistikovanější systémy, které jsou fyzicky na jednom místě, často používají pro transport dat speciální technologie (například InfiniBand), které zajišťují vysokou propustnost a nízkou dobu odezvy. Ostatní systémy, které mohou být i geograficky oddělené, používají Ethernet.

Distribuované systémy mají dnes velmi široké využití. Nachází uplatnění například v telefonních a počítačových sítích, distribuovaných databázích a síťových filesystémech, kontrolních a ovládacích systémech letadel, při vědeckých výpočtech či grafickém renderování. Pro mnoho zmíněných kategorií včetně vědeckých výpočtů je důležité disponovat velkým výpočetním výkonem a zároveň mít co nejnižší náklady. Sestavení clusteru z mnoha levných a úsporných jednotek by mohla být jedna z cest, jak tyto náklady minimalizovat.

Konstrukce výpočetního clusteru má mnoho výhod proti jednomu výkonnému multiprocesorovému počítači. Výpočetní cluster je výrazně spolehlivější, neboť výpadek jednoho uzlu neohrozí chod zbytku zařízení, lépe řeší problémy s odpadním teplem, protože více slabších procesorů se chladí snáz než jeden velmi výkonný a může být i ekonomicky příznivější, protože jednotlivé uzly mohou být různorodá levná zařízení. Tyto systémy lze škálovat na požadovaný výkon regulací počtu propojených jednotek a tím dosáhnout lepších parametrů než u většiny multiprocesorových stanic. Naopak mezi nevýhody patří náročnější údržba, režie při přenosu dat mezi uzly a vzájemné propojení jednotek přes výrazně pomalejší transportní médium, než jaké je k dispozici mezi procesory v jednom fyzickém počítači.

Cílem předkládané práce je zhodnocení vlastností malého jednodeskového počítače Raspberry Pi z hlediska jeho použití jako jednotky výpočetního clusteru. Budeme porovnávat výkonnostní i ekonomické (pořizovací cena, spotřeba elektrické energie) parametry celého řešení se zvoleným reprezentativním desktopovým a serverovým počítačem. Součást práce bude implementace programu, pomocí kterého budeme moci přibližně porovnat některé aspekty výpočetního výkonu testovaných počítačů. Zaměříme se na ty oblasti, které souvisí s použitím zařízení pro distribuované výpočty. Měřící program bude navržen a implementován se zohledněním možností Raspberry Pi, jako je například absence více procesorových jader. Měření bude provedeno na jednom kusu Raspberry Pi a referenčním desktopovém i serverovém počítači.

V kapitole 1 představíme Raspberry Pi, zajímavé příklady využití a některé konkurenční jednodeskové počítače. V kapitole 2 stručně shrneme poznatky o architektuře počítačů, uvedeme několik známějších programů určených k porovnávání výkonu a definujeme naše požadavky na takovýto vhodný program.

V kapitole 3 se zaměříme na návrh vlastní měřící aplikace včetně volby testovacích algoritmů. Technické detaily implementace a ovládání vytvořeného měřícího programu popíšeme v kapitole 4. Kapitola 5 bude pojednávat o metodice měření, získaných hodnotách, jejich zpracování a zhodnocení. V závěru shrneme zjištěné údaje a vyslovíme doporučení pro stavbu a provoz výpočetního clusteru z Raspberry Pi.

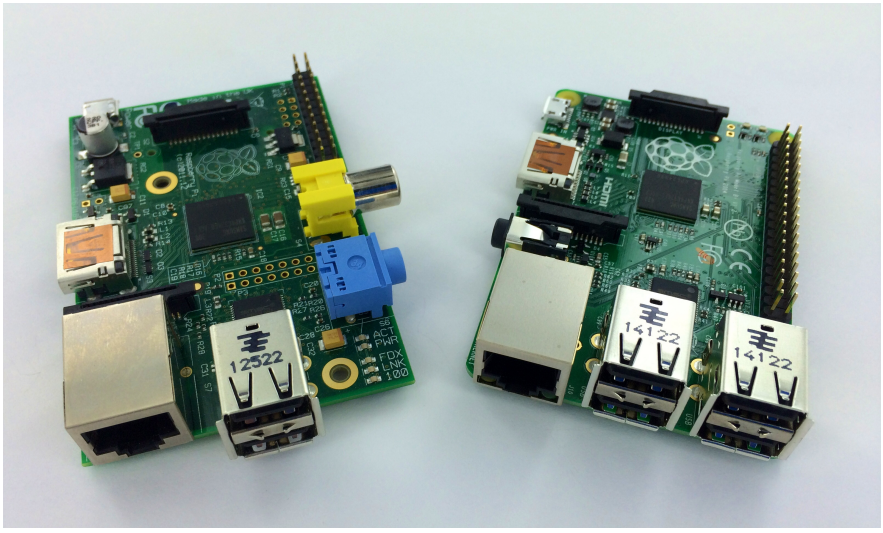
1. Raspberry Pi

Raspberry Pi [1] je jednodeskový počítač velikosti odpovídající kreditní kartě. Vývoj tohoto počítače zajišťuje britská nadace Raspberry Pi Foundation. Projekt levného počítače vznikl díky snaze autorů zlepšit výuku programování na školách. Dnes má mnoho dalších využití, například jako domácí webový server, řídicí jednotka různých elektronických zařízení nebo plnohodnotný počítač pro nenáročné uživatele.

Myšlenka na vytvoření malého počítače sloužícího k výuce programování pro děti pochází z roku 2006 od zaměstnanců univerzity v Cambridge, kteří byli znepokojeni snižující se úrovní programátorských dovedností u nově nastupujících studentů. Chtěli navrhnout počítač, který by byl dostatečně levný a na kterém by si kdokoli mohl vyzkoušet programování.

Raspberry Pi se začalo vyrábět ve třech verzích – model A, model B a samostatný výpočetní modul. Na začátku roku 2012 byl oficiálně zahájen prodej počítače Raspberry Pi model B s cenou 35 $. Ostatní varianty se dočkaly uvedení na trh později.

Dva roky po spuštění prodeje překročil celkový počet prodaných kusů Raspberry Pi 3 miliony. V létě téhož roku byl představen nový model B+. Cenová politika zůstala stejná jako u původního modelu B. Hlavní novinkou je optimalizovaná spotřeba energie. Vzhled obou modelů je na obrázku 1.1.



Obrázek 1.1: Porovnání modelu B (vlevo) a novějšího B+ (vpravo). *Zdroj: One Mans Anthology*

Z hlediska stavby výpočetního clusteru je Raspberry Pi vhodné především díky nízké pořizovací ceně, malé spotřebě elektrické energie a zároveň dostatečnému výpočetnímu výkonu pro běh operačního systému. Výhodou jsou malé rozměry a pasivní chlazení celé desky. Další pozitivní faktory jsou aktivní vývoj a široká podpora ze strany výrobce i komunity.

1.1 Technické parametry

Specifikace modelů B revize 2.0 a B+ [2]:

- procesor Broadcom BCM2835 (ARMv6) s taktem 700 MHz, GPU VideoCore IV s taktem 250 MHz,
- 512 MB paměti DDR2 SDRAM sdílených s GPU,
- 10/100 Mbit/s Ethernet adaptér s konektorem RJ45, vnitřně připojený přes USB,
- konektor pro SD kartu (microSD u B+) – persistentní úložiště,
- HDMI (a Composite u modelu B) obrazový výstupní port,
- dva USB 2.0 porty (čtyři u B+),
- zvukový výstup přes 3,5 mm konektor,
- další specifické porty pro přídavné obvody.

Počítač se napájí pomocí microUSB portu zdrojem se stejnosměrným napětím 5 V. Spotřeba elektrické energie se podle specifikace pohybuje okolo 3,5 W (závisí na připojených periferiích), u novějšího modelu je až o 1 W nižší.

Raspberry Pi Foundation vyvíjí a podporuje vývoj nového příslušenství. Mezi populární patří Camera module, přídavný fotoaparát s funkcí HD kamery pro Raspberry Pi. Velmi podobný je Pi NoIR, což je modul fotoaparátu bez infračerveného filtru umožňující noční vidění [1].

Rozšiřující desky nabízí vylepšení základního modelu počítače. Obsahují další konektory, LED diody, přídavnou paměť, A/D převodníky a další. Populární jsou UniPi Board [3] a Gertboard [1].

S novým modelem B+ přišly vylepšené rozšiřující desky označované anglickým termínem HAT – Hardware Attached on Top [4]. Tyto desky využívají přidaných GPIO pinů počítače k lepší spolupráci se systémem už od bootovacího procesu.

1.2 Operační systémy a software

Na oficiálních stránkách Raspbery Pi Foundation [1] jsou k dispozici odkazy na stažení obrazů různých systémů a distribucí. Většina z nich vznikla z běžných desktopových distribucí GNU/Linuxu. Oficiálně je doporučován Raspbian [5]. Jedná se o svobodný operační systém založený na Debianu optimalizovaný pro hardware Raspberry Pi. Pro měření v této práci budeme používat verzi 2014-12-24 založenou na Debianu Wheezy.

Další rozšířené linuxové systémy jsou Pidora (Fedora Remix) a Arch Linux ARM. Multimediální centra zastupují Raspbmc a OpenELEC, oba založené na XBMC open-source multimediálním přehrávači (dnes známým pod názvem Kodi).

Poslední rozšířenější systém je RISC OS, který byl již při svém vzniku v roce 1987 určen výhradně pro procesory s architekturou ARM. Mezi dalšími funkčními systémy můžeme zmínit Android, openSUSE, Slackware ARM, FreeBSD, OpenWrt, Kali Linux, Kano OS a Ark OS [6].

Někteří výrobci proprietárních aplikací uvolnili své programy pro Raspberry Pi zdarma. Součástí standardní instalace Raspbianu je také programovací jazyk Wolfram Language a matematický program Wolfram Mathematica [7], aktuálně ve verzi Mathematica 10. Další podobný projekt je Minecraft Pi Edition [8].

Od raného stádia vývoje počítače Raspberry Pi byl o tento projekt velký zájem. Jednalo se o první cenově dostupný plnohodnotný počítač s velmi malými rozměry, pasivním chlazením a příznivou spotřebou. Brzy se kolem něj vytvořila poměrně rozsáhlá komunita lidí. Práce těchto lidí je velmi vítána a oceňována, ať už jde o samotný vývoj softwaru či jako komunitní technická podpora [1].

1.3 Příklady využití

Raspberry Pi se používá v mnoha zajímavých projektech a zařízeních. Zde uvedeme jen pár vybraných příkladů, které dostatečně demonstrují možnosti tohoto počítače.

Přístroj na měření radiace Ionizující záření je pro člověka ve větší míře nebezpečné. Jeho intenzita se měří Geiger-Müllerovým počítačem. Zájem o tyto přístroje stoupl po jaderné havárii v elektrárně Fukušima I v Japonsku. Raspberry Pi je pro tento typ přístroje velmi vhodné. Je malé, levné, umožňuje připojit různá výstupní zařízení (monitor, reproduktory, vzdálený přijímač na síti), má nízkou spotřebu elektrické energie. K vytvoření je potřeba kromě počítače Raspberry Pi také přídavný modul PiGI a Geiger-Müllerova trubice [9].

Raspberry Pi ve stratosféře Vypouštění balónů s měřícími zařízeními do vysokých výšek blíže k okraji vesmíru se stává čím dál častější. Mikropočítač Raspberry Pi zde posloužil k pořizování fotografií ve výšce necelých 40 km nad povrchem a jejich okamžitému odesílání zpět na Zemi. Nahradil Arduino Mini Pro, které přes nižší hmotnost a spotřebu energie nedokázalo komunikovat s webkamerou. Mikropočítač vydržel pracovat i při teplotách okolo –50°C a úspěšně přistál zpět [10].

Meteorologické stanice Malé meteorologické stanice jsou nedílnou součástí mnoha domácností. K Raspberry Pi lze připojit vlastní měřící modul AirPi [11] nebo některou z podporovaných konvenčních stanic [12]. S připojením k internetu lze sledovat naměřené hodnoty odkudkoli. Výhody Raspberry Pi jsou malá velikost, příznivá spotřeba energie a podpora ze strany některých výrobců meteorologických stanic a vývojářů obslužného softwaru.

Fotografování S Raspberry Pi lze díky přídavnému fotografickému modulu vytvářet zajímavé fotografie či videa. Populární jsou videa zachycující jedno místo v různých časech (time-lapse) a videa s efektem letící kulky (bullet time effect). Ta jsou složena z fotografií scény v jediném okamžiku z různých úhlů [13]. Tyto efekty byly poprvé ve větší míře využívány ve filmech Matrix. Raspberry Pi je vhodné pro tyto účely zejména díky nízké pořizovací ceně (je nutné mít větší počet fotografických zařízení) a schopnosti propojení (je nutné dát všem fotoaparátům povel ve stejný čas), například přes ethernetový port. Velkou výhodou je přídavný

fotoaparát, který je velmi malý, má dostatečné rozlišení a je pro Raspberry Pi dobře dostupný a podporovaný.

Kvadrokoptéra Zařízení schopné letu, které je poháněné čtyřmi nezávislými rotory, lze ovládat také pomocí Raspberry Pi. To spolupracovalo se zařízením Arduino, které obstarávalo ovládání periferií. Raspberry Pi pak provádělo výpočty stabilizace v reálném čase. Tento počítač byl zvolen z důvodu možnosti připojení mobilního 3G modemu, webové kamery a schopnosti dostatečně rychle provádět výpočty nutné k řízení letu [14].

Z výše uvedených příkladů je vidět, že Raspberry Pi je osvědčené zařízení využívané především v situacích, kdy je významným faktorem nízká energetická náročnost nebo nutnost většího počtu spolupracujících jednotek. To dává dobrý příslib pro toto zařízení při použití jako uzel výpočetního clusteru, kde jsou obě tyto vlastnosti důležité. Ukázalo se, že mikropočítač dokáže pracovat i v náročnějších podmínkách, což také umožňuje jeho využití v robustních systémech, které musí pracovat v terénu.

1.4 Podobná zařízení

Dnes existuje více druhů jednodeskových počítačů, Raspberry Pi byl však první, který se dostal do podvědomí širšího okruhu lidí a stal se velmi populární. Tím výrazně podpořil celý trh s jednodeskovými počítači a mnohá zařízení mají stále Raspberry Pi jako svůj vzor, ze kterého vychází nebo se jím inspirovala. Na dnešním trhu existují mikropočítače s různým výkonem, ale také různou cenou. Raspberry Pi nepatří mezi nejvýkonnější zařízení, ale příznivým poměrem ceny a výkonu se řadí mezi nejvýhodnější. Pro srovnání jsme vybrali několik známých konkurenčních zařízení[1].

AMD Gizmo 2 [15] patří k hardwarově vybavenějším jednodeskovým počítačům. Použitý procesor podporuje instrukční sadu kompatibilní s x86 a deska obsahuje 1 GB paměti RAM. K dispozici je 8 portů USB, port mSATA, HDMI, 3.5 mm zvukový výstup a sada GPIO pinů. Nevýhodou je aktivní chlazení pomocí ventilátoru, vyšší pořizovací cena a zhruba třikrát vyšší spotřeba energie oproti Raspberry Pi. Pokud bychom umístili více těchto zařízení do omezeného prostoru, bylo by především nutné vyřešit problém s odvodem přebytečného tepla.

Arduino Uno [16] je malý jednodeskový počítač s mikrokontrolerem ATmega328, 2 kB SRAM a 32 kB flash pamětí. Tento hardware neumožňuje běh operačního systému, deska obsahuje pouze zavaděč pro spouštění nahraného programu. Podporovaný programovací jazyk vychází z C a C++, ale v některých ohledech se může mírně lišit. Existuje více modelů desek Arduino, ale žádná nedosahuje výpočetního výkonu Raspberry Pi. Jako uzel ve výpočetním clusteru je toto zařízení nevhodné kvůli svému nízkému výkonu, nutnosti použít specifický programovací

[1] Porovnání jednodeskových počítačů je například na anglické wikipedii (`http://en.wikipedia.org/wiki/Comparison_of_single-board_computers`) nebo na stránkách `http://iqjar.com/jar/an-overview-and-comparison-of-todays-single-board-micro-computers/`.

jazyk a problematickému síťování pomocí pinů na desce.

Arndale Board [17] pohání procesor Samsung Exynos 5, který spolupracuje s 2 GB paměti RAM. Deska je osazena některými nadstandardními komponenty, obsahuje například i port USB 3.0. Výrobce také nabízí řadu příslušenství jako 7-palcový LCD display, Wi-Fi, Bluetooth a GPS modul. Cena je výrazně vyšší než u Raspberry Pi, avšak toto zařízení by také mohlo být vhodné jako jednotka pro výpočetní cluster.

Projekt **Banana Pi** [18] vznikl odštěpením od projektu Raspberry Pi. Cílem bylo poskytnout zákazníkům lepší hardware, ale zachovat kompatilibitu s původním Raspberry Pi v nejvyšší možné míře. Banana Pi obsahuje procesor ARM Cortex-A7 dual core, 1 GB RAM a gigabitový ethernetový port. Nejběžnější operační systémy jsou Debian, Raspbian, Arch Linux a Android. Cena zůstala příznivě nízká, ale je vyšší než u Raspberry Pi. To vše dělá z Banana Pi vhodného alternativního kandidáta pro stavbu clusteru.

BeagleBoard Black [19] má 1 GHz procesor ARM Cortex-A8, 512 MB paměti RAM a 4 GB interní flash paměti. Hlavní operační systémy jsou Ubuntu, Android a Debian. Oproti Raspberry Pi není BeagleBoard tolik rozšířený, má vyšší spotřebu energie, je o 10 $ dražší, ale je nepatrně výkonější. Nevýhodou je pouze 4 GB vestavěné paměti bez jiné možnosti rozšíření než dalšího paměťového média zapojeného přes USB rozhraní, což z hlediska stavby výpočetního clusteru může být významné omezení.

Intel Edison [20] je nová vývojová platforma určená převážně pro nositelnou elektroniku, kterou představila na konferenci CES 2014 společnost Intel. Jedná se o počítač velikost SD karty, který obsahuje dvoujádrový procesor a integrovaný Wi-Fi a Bluetooth modul. Bylo oznámeno, že použitým operačním systémem bude Linux. Cluster sestavený z desek Intel Edison by mohl být velmi malý, ale pouze bezdrátová konektivita by mohla být při větším počtu zařízení úzké místo celého řešení.

Intel Galileo [21] je vývojová deska, která je inspirována zařízeními Arduino. Zachovává i některá rozhraní podle desky Arduino Uno. Obsahuje jednojádrový 32bitový procesor Intel Quark SoC X1000 s instrukční sadou kompatibilní s x86, 256 MB DDR3 RAM. Firmware počítače je Yocto 1.4 Poky Linux. Pro programování se používá originální Arduino IDE. Deska nedosahuje výkonu Raspberry Pi a neumožňuje běh plnohodnotného operačního systému. Z důvodů slabšího výkonu a přibližně dvojnásobné ceně proti Raspberry Pi je toho zařízení méně vhodné pro stavbu výpočetního clusteru.

Parallella [22] je jednodeskový počítač zaměřený na paralelní výpočty. Obsahuje procesor Zynq-7000 Series Dual-core ARM A9, akcelerátor Epiphany s 16 nebo 64 jádry, 1 GB RAM, gigabitový ethernetový port, HDMI výstup a další konektory. Dodává se s předinstalovaným Ubuntu OS. Pro optimální využití jsou doporučovány programovací jazyky C a C++. Zařízení je hardwarově lépe vybavené než Raspberry Pi, ale podle konkrétní specifikace je 3 až 6 krát draž-

ší. Parallella by mohla být vhodné alternativní zařízení pro stavbu výpočetního clusteru, vyšší cena může být vyvážena odpovídajícím výkonem.

V této práci se budeme detailněji zabývat pouze Raspberry Pi model B revize 2.0 a model B+, ale dále navržená měření výkonu lze použít i na jiná zařízení splňující základní požadavky na běh operačního systému GNU/Linux. Z výše zmíněných by bylo vhodné otestovat zařízení Arndale Board, Banana Pi a Parallella. Měření na těchto dalších zařízeních je již mimo rozsah této práce, ale s využitím připravených nástrojů by ho bylo možné snadno provést.

1.5 Využití více Raspberry Pi

Myšlenka stavby clusteru, jehož jednotky tvoří Raspberry Pi, byla již několikrát realizována. Velmi často bylo cílem sestrojit levné a dostupné zařízení, které se svou architekturou přibližuje moderním superpočítačům, především pro účely výuky, studia jeho vlastností a ladění distribuovaných algoritmů. Některé projekty nalezly i praktické uplatnění, např. jako webový server s rozložením zátěže na jednotlivé uzly. Jiným cílem může být konstrukce robustního zařízení, protože velké množství malých zařízení může být v tomto ohledu výhodnější než jeden výkonný server.

Příkladem může být cluster sestavený J. Kiepertem na Boise State University z 32 jednotek [23]. Jeho projekt vznikl jako nástroj pro disertační práci, protože z důvodu oprav nemohl využívat univerzitní cluster. V publikaci se kromě základních výkonnostních charakteristik poměrně podrobně věnuje i mechanické konstrukci a řešení napájení a chlazení.

Na univerzitě v Southamptonu ve Velké Británii postavili v roce 2012 zařízení Iridis-Pi [24], které se skládá z 64 jednotek Raspberry Pi model B. Použili optimalizovaný operační systém Raspbian a podrobně popsali softwarovou přípravu jednotlivých jednotek. Benchmarkem *LINPACK* měřili výkonnost jednotlivých uzlů, celý cluster testovali pomocí *HPL* (*High Performance LINPACK*). Při řešení soustavy lineárních rovnic $Ax = b$ řádu 10 240 dosáhli celkového výsledku 1,14 GFLOPS. Studovali výkonnostní charakteristiky v závislosti na velikosti úlohy i na počtu aktivních uzlů, věnovali se i hodnocení síťového připojení a přístupu na SD kartu. Kromě výuky vidí možné využití podobných zařízení díky své robustnosti a nízké spotřebě např. v automobilovém průmyslu či ve vojenské oblasti.

Stavba výpočetního clusteru vyžaduje určité technické znalosti a finanční prostředky, proto je vhodné mít předem podklady o předpokládaném výkonu a ekonomické stránce projektu. Je třeba vzít v úvahu mimo jiné odvod přebytečného tepla, zajištění dostatečného napájení a vhodné propojení jednotlivých zařízení. Protože nemáme k dispozici velké množství Raspberry Pi, provedeme měření výkonu a spotřeby na jednom zařízení a z nich odhadneme výsledné parametry clusteru postaveného z těchto jednotek. Získané hodnoty následně porovnáme s údaji naměřenými na Iridis-Pi. Připravená testovací aplikace na porovnávaní výkonu počítačů je snadno přenositelná i na jiná zařízení, takže lze snadno provést měření i na ostatních jednodeskových počítačích, které jsme dříve doporučili.

2. Výpočetní výkon

Základní předpoklad pro výběr nejvhodnějšího zařízení je schopnost jejich porovnání. To lze dělat mnoha způsoby podle toho, které parametry produktu upřednostňujeme. Mohou to být výkon, pořizovací cena, provozní náklady, vzhled, výrobce a další. V oblasti počítačového hardwaru nás zajímají především první tři. Nejproblematičtějším bodem je samotné měření výpočetního výkonu, které závisí na konkrétní architektuře počítače a požadavcích zadavatele. Jako měřící nástroj se používá sada testů (algoritmů nebo počítačových programů), které v dostatečné míře implementují zadané požadavky. Taková sada se obecně nazývá počítačový benchmark. Výsledky, které vypovídají o výpočetním výkonu, mohou být nejčastěji doba běhu testu, velikost zpracovaných dat nebo latence či propustnost. Tyto výsledky jsou vždy závislé na konkrétním druhu testování a nelze je porovnávat mezi různými benchmarky. Většinou bývá nutné dodatečné statistické zpracování naměřených údajů, případně odfiltrování odlehlých hodnot.

2.1 Architektura počítačů

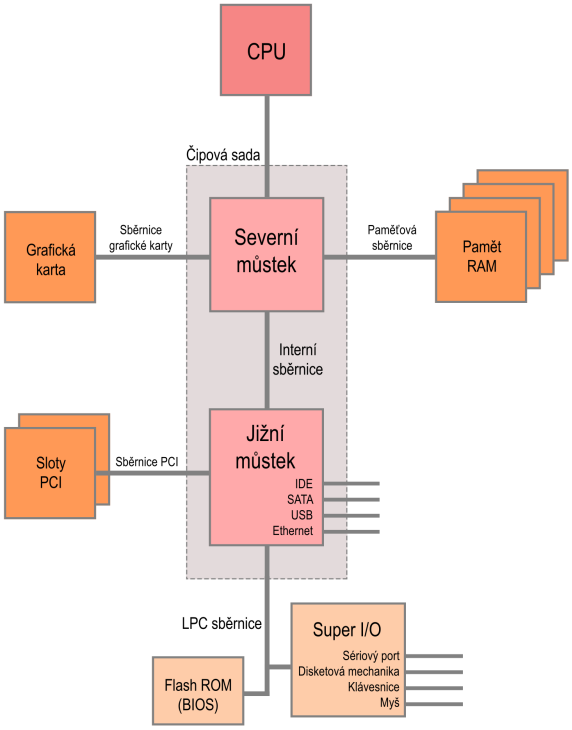
Architektura dnešních počítačů [25] je poměrně složitá a do značné míry různorodá. V této kapitole přiblížíme základní součásti počítače, jejich funkci a vzájemné propojení. Nebudeme se zabývat detailním popisem všech komponent moderních počítačů, ale zaměříme se pouze na činnost částí, které mají největší vliv na celkový výpočetní výkon. Základní schéma propojení jednotlivých komponent je na obr. 2.1.

Hlavní částí počítače je procesor. Ten úzce spolupracuje s čipovou sadou, která se skládá z jednoho nebo více integrovaných obvodů, jejichž funkce je zprostředkovávat komunikaci mezi procesorem, sběrnicemi, řadiči a dalšími součástkami. Termín čipová sada obvykle označuje dva čipy – severní můstek (northbridge) a jižní můstek (southbridge). Obě části spolu komunikují po interní systémové sběrnici. Severní můstek se stará o komunikaci mezi procesorem a operační pamětí, případně i grafickým čipem. Jižní můstek integruje rozšiřující sběrnice (PCIe, SATA, ...), skrze které se připojují periferní zařízení. Nejnovější trend vývoje je umisťování těchto součástek přímo do procesoru.

2.1.1 Procesor

Procesor neboli CPU (Central Processing Unit) je integrovaný obvod provádějící strojové instrukce. Různé procesory mohou zpracovávat různý strojový kód. Množina platných instrukcí pro procesor (jeho instrukční sada) je dána jeho architekturou. Způsob, jakým je v procesoru implementována daná instrukční sada, nazýváme mikroarchitektura. Pro jednu architekturu může existovat více mikroarchitektur, které ji implementují. Mezi architektury procesorů patří například ARMv6KZ (Raspberry Pi) či Intel 64 (počítače se kterými budeme Raspberry Pi porovnávat, jejich použitá mikroarchitektura je Nehalem).

Na obr. 2.2 je schéma procesoru Intel Core i7 Nehalem [26]. Jedná se o jeden z procesorů, na kterých budeme provádět měření. Dnešní procesory Intel s mikroarchitekturou Haswell se v základních principech fungování neliší.



Obrázek 2.1: Schéma architektury počítače.

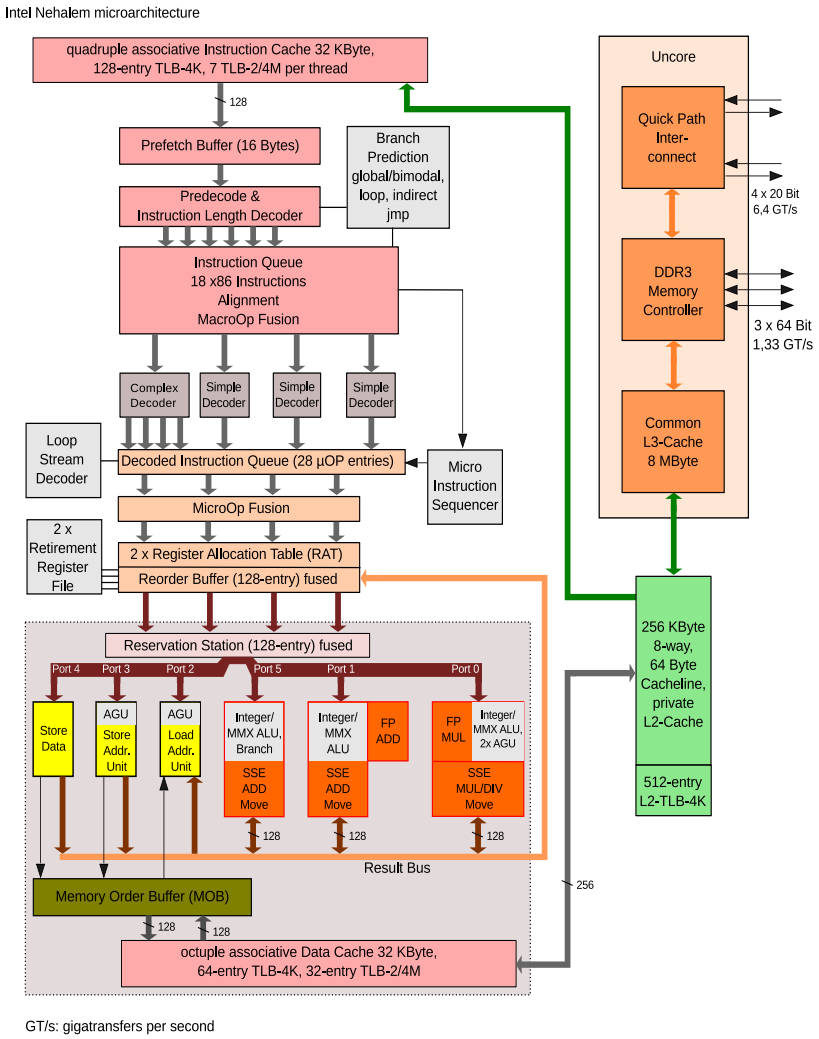
Procesor může obsahovat jedno nebo více výpočetních jader. Většina procesorů z rodiny Intel Core i7 obsahuje čtyři fyzická jádra podporující Hyper-Threading, díky čemuž se procesor jeví jako osmijádrový. Kvůli tomuto rozdělení je zajištěno lepší využití každého fyzického jádra.

Procesor při svém běhu čte jednotlivé instrukce, které dále zpracovává odpovídajícím způsobem. Většina procesorů pro urychlení výpočtu používá pipelining, díky kterému je v jednom okamžiku rozpracováno více instrukcí, což přináší lepší využití procesorových obvodů.

Rychlost procesoru je nejčastěji udávána pomocí taktovací frekvence. Každá instrukce se provede za předem stanovený počet taktů, ale kvůli různým technikám zvýšení výkonu v dnešních procesorech (například pipelining) již frekvence nemusí být rozhodujícím faktorem výpočetního výkonu pro porovnávání procesorů, přesto jeho rychlost stále významně ovlivňuje výkon celého počítače. Parametry procesoru se nejvíce projeví u výpočetně náročných úloh, které pracují s omezenou množinou dat.

2.1.2 Operační paměť

Operační paměť se běžně označuje zkratkou RAM (Random Access Memory), která původně označovala jakoukoli paměť s náhodným přístupem k datům. Polovodičové paměti RAM rozdělujeme podle technologie uchovávání informace na statickou (SRAM) a dynamickou (DRAM). Statické paměti jsou velmi rychlé, ale díky své konstrukci také drahé. Používají se např. ve vyrovnávacích pamětech procesorů. Dynamické paměti jsou levnější a jednodušší, ale potřebují po určitém



Obrázek 2.2: Schéma procesoru Intel Core i7. *Zdroj: Wikipedia*

čase obnovovat obsah každé paměťové buňky. Tento typ je používán jako operační paměť v osobních i serverových počítačích.

Operační paměť slouží k uložení kódu programu, vstupních i výstupních dat i průběžných mezivýsledků výpočtu. Protože procesor je výrazně rychlejší než paměť RAM a přímá práce s touto pamětí by ho zdržovala, používají se jako mezistupeň rychlejší asociativní vyrovnávací paměti.

Asociativní vyrovnávací paměť (cache) slouží jako vyrovnávající mezivrstva mezi dvěma subsystémy pro zvýšení celkového výkonu. V procesoru cache zajišťují rychlý přístup k často používaným datům. Představený procesor Intel Core i7 obsahuje tři úrovně vyrovnávacích pamětí – L1, L2 a L3. Se vzrůstající úrovní paměti roste její kapacita, ale také klesá rychlost. První dvě úrovně má každé jádro vlastní, L3 je sdílená všemi procesorovými jádry. Hierarchie pamětí cache je znázorněna na obr. 2.2. Tyto paměti jsou velmi důležité pro výkon celého počítače, protože omezují čekání procesoru na dokončení operací načítání a ukládání dat.

Pro správu paměti RAM v operačním systému se používá koncept stránkování. Každý běžící proces má vlastní souvislou virtuální paměť, která se podle potře-

by mapuje do fyzické operační paměti. Toto mapování se provádí po souvislých částech dané velikosti (typicky 4 kB), které nazýváme paměťové stránky. Převod virtuální adresy na adresu korespondující stránky ve fyzické paměti zajišťuje jednotka MMU (Memory Management Unit), která bývá součástí procesoru. Protože k překladům adres mezi virtuální a fyzickou pamětí dochází velmi často a jedna operace překladu znamená i několik čtení z paměti, existuje v procesoru speciální TLB cache (Translation Lookaside Buffer), která urychluje čtení hodnot ze stránkovací tabulky a tím výrazně zvyšuje rychlost překladu adres.

Parametry paměti RAM mohou u některých úloh výrazně ovlivňovat výkon celého počítače. Důležitá je jak její rychlost, tak také dostatečná kapacita. Rychlost konkrétní aplikace ve velké míře závisí na jejím způsobu práce s daty, především zda lze v plné míře využít procesorové paměti cache.

2.1.3 Periferní zařízení

K interakci s počítačem jsou nezbytnou součástí různá vstupní a výstupní zařízení, souhrnně označovaná jako periferní zařízení. Může se jednat o grafický čip s připojeným monitorem, paměťové médium, tiskárnu, klávesnici a myš, webkameru a mnohá další.

Pro výpočetní cluster z jednotek Raspberry Pi je nejdůležitější persistentní paměťové úložiště a Ethernet. Přístup do paměťového úložiště se používá v každé aplikaci (minimálně při jejím spuštění), síť používá cluster z Raspberry Pi jako komunikační médium. Ostatní periferní zařízení mají v této konfiguraci jen velmi malé uplatnění.

Nejpoužívanější persistentní úložiště pro desktopové a serverové počítače jsou HDD (Hard Disk Drive) a SSD (Solid State Drive) disky. HDD je médium s několika rotačními plotnami, kde pohyblivá hlavička čte či zapisuje magnetickou stopu. Tento typ disků dnes umožňuje uchovávat data v jednotkách terabajtů. Nevýhodou je nízká odolnost při otřesech v průběhu čtení a zápisu. Alternativní SSD disky jsou založeny na technologii flash pamětí, které uchovávají informace pomocí soustavy tranzistorů. Tyto disky dosahují vyšších rychlostí čtení i zápisu, ale jejich pořizovací cena je výrazně vyšší proti klasickým HDD. Testovaný referenční desktopový počítač používá SSD disky a k serverovému počítači je připojeno diskové pole s oběma uvedenými typy disků. Nejčastější připojení těchto periferních zařízení je přes SATA rozhraní do jižního můstku, případně vzdáleně přes počítačovou síť.

Pro mobilní zařízení a jednodeskové počítače včetně Raspberry Pi jsou nejpoužívanější SD karty (případně microSD), které obsahují flash paměť podobně jako SSD disky. Hlavní předností proti klasickým HDD diskům je jejich malá fyzická velikost, proto se často používají v drobných zařízeních jako jediné paměťové úložiště. S velikostí pak úzce souvisí také jejich řádově nižší kapacita i přenosová rychlost.

Druhé podstatné periferní zařízení pro uzel v clusteru je Ethernet, souhrn technologií pro počítačové sítě. Aktuální standardizovaná verze je schopna přenosové rychlosti až 100 Gbit/s, avšak běžná zařízení podporují rychlosti 100 Mbit/s a 1 Gbit/s. Jako přenosové médium se používají kabely s páry kroucených vodičů nebo optické kabely. Podstatné parametry týkající se výkonu jsou rychlost výměny dat a zpoždění způsobené přenosem. Cluster z jednotek Raspberry Pi bude

využívat Ethernet jako komunikační médium z důvodu jeho rozšířenosti a podpory zařízením. Alternativní možnost propojení, která by se vyrovnala výkonu Ethernetu, deska nenabízí.

Grafický čip by mohl být využit pro jisté typy výpočtů, ale bohužel většina jednodeskových zařízení nepodporuje žádné standardní rozhraní (jako OpenCL nebo Nvidia CUDA). V této oblasti je zatím nevyužitý potenciál těchto malých zařízení, protože grafické čipy se svým velkým množstvím výpočetních jader jsou velmi výkonné pro datově-paralelní druhy výpočtů.

2.2 Měření počítačového výkonu

K měření výkonu počítačů [27] lze použít počítačový benchmark. Jedná se o experimentální metodu s omezenou přesností, jejímž cílem je relativní porovnání několika počítačů. Měření probíhá nejčastěji tak, že na cílovém počítači spouštíme vybrané algoritmy a zaznamenáváme dobu jejich běhu. Vždy je důležité správně zhodnotit aktuální požadavky a najít vhodné testovací algoritmy a odpovídající vstupní data.

Benchmarky lze dělit do několika skupin podle své funkce, ale neexistuje žádná přesná a všeobecně přijímaná definice takového dělení. Tyto skupiny navíc nejsou vždy disjunktní a přesně vymezené. Některé benchmarky mají více funkcí a proto je můžeme zařadit do více skupin současně. Nejčastěji rozlišujeme tyto typy:

- aplikační benchmark,
- mikrobenchmark,
- kernel benchmark,
- syntetický benchmark,
- I/O (input/output, vstupně-výstupní) benchmark.

Aplikační benchmark měří výkon reálného programu, mikrobenchmark se zabývá testováním jednotlivých komponent systému, převážně se skládá z relativně malých a specificky zaměřených kusů kódu. Kernel benchmark měří základní architektonické rysy paralelních počítačů. Syntetické benchmarky zohledňují četnosti jednotlivých operací získané z mnoha aplikačních programů a testují výkon počítače na programu napsaném podle získaných četností. I/O benchmarky testují periferní zařízení.

Vhodný benchmark pro celkové změření výkonu jednodeskových počítačů nelze jednoduše zařadit. Měl by testovat jednotlivé komponenty a periferní zařízení počítače odděleně i celkové chování systému na reálně používaných algoritmech.

2.2.1 Vybrané benchmarky na platformě Linux

Jako operační systém pro Raspberry Pi v zamýšleném použití použijeme některou z distribucí GNU/Linuxu. To je jediný systém, který má dostatečnou podporu většiny malých zařízení a vyhovuje pro použití ve výpočetním clusteru. Proto se v následujícím zaměříme pouze na produkty pro tento operační systém, případně produkty multiplatformní.

Phoronix Test Suite [28] je open-source benchmark, který je poměrně rozšířený a velmi rozsáhlý. Podporuje širokou škálu operačních systémů a architektur procesorů. Obsahuje přes sto testovacích modulů s možností rozšiřování. Celý projekt je napsán v jazyce php a je dobře zdokumentován. Systém umožňuje opakované spouštění testů, nahrávání výsledků na web *OpenBenchmarking.org*, grafické zobrazení naměřených výsledků, sledování relevantních údajů ze systémových logů, sledování sensorů v průběhu testování a další. Výhodou je široká komunita okolo tohoto projektu a finanční podpora ze strany komerčních organizací.

Bonnie++ [29] je UNIXový benchmark zaměřený na souborový systém a diskové operace. Vznikl v roce 1999 z původního projektu *Bonnie* úpravou a přepsáním do jazyka C++. Výstup ukládá do souboru v CSV formátu vhodném ke strojovému zpracování. Vývoj probíhá velmi pomalu.

Benchmarky **SPEC** (od *Standard Performance Evaluation Corporation*) [30] jsou velmi známé a populární. Nejznámější je sada zaměřená na výkon procesoru, *SPEC CPU2006*. Tyto benchmarky se vyznačují vysokou přesností a ve všech verzích se snaží zachovávat stejné standardy. Většina produktů organizace *SPEC* je dostupná pouze komerčně za ceny od 50 $ do 5000 $, proto pro účely této práce nevyhovuje.

LMBench [31] je známý a dlouhodobě vyvíjený UNIXový benchmark. *LMBench* se zabývá měřením latence a propustnosti v několika různých oblastech, například propustnost systémové roury (pipe) či latence přepínání kontextu. *LMBench* je napsaný v jazyce ANSI C.

LINPACK [32] je původně knihovna pro výpočty týkající se lineární algebry napsaná v jazyce Fortran. *LINPACK benchmark* byl poprvé uveden v roce 1979 jako příloha k uživatelskému manuálu dodávanému ke knihovně. Benchmark je zaměřen na operace s čísly v plovoucí desetinné čárce. Provádí řešení soustavy n lineárních rovnic $Ax = b$. Jeho paralelní verze *HPL* (*High Performance LINPACK*) se používá k testování paralelních počítačů a distribuovaných systémů. Výsledky měření *HPL* jsou mimo jiné použity i k sestavování žebříčku 500 nejlepších superpočítačů planety [33]. Jsou k dispozici oficiální implementace v jazycích Java, C a Fortran.

Každý z výše prezentovaných programů nějakým způsobem pro naše účely nevyhovuje. Některé používají interpretovaný programovací jazyk, nepodporují všechny požadované testy či jsou dostupné pouze pod komerční licencí.

2.3 Přehled požadavků

Hlavním cílem práce je přinést porovnání výkonu a ceny (pořizovací i provozní) clusteru postaveného z mnoha jednotek jednodeskových počítačů oproti běžně dostupným serverovým a desktopovým počítačům. Motivací je zpracovávání dobře paralelizovatelných úloh v rozumném čase a s co nejmenšími náklady.

V této sekci stanovíme základní parametry, které by měl splňovat vhodný

benchmark. Chceme provést testy jednotlivých komponent zvlášť i jejich kombinací na reálně používaných algoritmech. Zajímá nás především procesor, operační paměť, persistentní úložiště a síťový subsystém. Tento výběr plyne z plánovaného použití jednodeskového počítače jako jednotky ve výpočetním clusteru, kde každý uzel bude zapojen v konfiguraci pouze s připojenou sítí a bude se podílet na zpracování výpočetně náročných úloh.

- Procesor
  - výkon aritmetických operací s celými čísly a čísly s plovoucí desetinnou čárkou
  - využití procesorových pamětí cache
- Paměť RAM
  - rychlost čtení a zápisu
  - rychlost přístupu v závislosti na využití pamětí cache
- Persistentní úložiště
  - rychlost sekvenčního a náhodného čtení ze souboru a sekvenčního zápisu do souboru
- Ethernet
  - latence a rychlost TCP i UDP přenosu po různě velkých blocích

Z důvodu použití jednojádrového procesoru v počítači Raspberry Pi budou všechny testy testy navrženy jako jednovláknové.

Dále bychom od implementace měřícího programu chtěli dodržení následujících podmínek.

- Benchmark by měl jít spustit na operačním systému GNU/Linux. Důvodem je podpora mnoha typů zařízení včetně většiny jednodeskových počítačů, možnost vlastní úpravy a bezplatná distribuce.
- Testovací program by měl podporovat minimálně dvě architektury procesorů, ARM a x86 (IA-32, případně Intel 64). To jsou dnes nejrozšířenější architektury, které pokryjí většinu jednodeskových, serverových i desktopových počítačů.
- Benchmark půjde přeložit do nativního kódu dané platformy z důvodu omezených prostředků dnešních jednodeskových počítačů. Jazyky používající správu paměti pomocí *garbage collectoru* nejsou vhodné, protože programátor nemá možnost sám uvolňovat naalokovanou paměť a spuštění úklidu paměti v nevhodnou dobu může výrazně ovlivnit výsledky měření.
- Testovací aplikace by měla pokrýt všechny požadované testy a měla by zachovávat jednotné uživatelské rozhraní, které usnadní orientaci v programu a především strojové zpracovávání výsledků. Vzhledem k předpokládanému použití programu počítačově zdatnými uživateli je konzolové uživatelské rozhraní naprosto dostatečné.

- Benchmark by měl být k dispozici zdarma včetně zdrojových kódů jako open-source, aby bylo možné ho udržovat a dále rozšiřovat.

Z výše uvedených požadavků vyplývá, že žádný z dříve zmiňovaných benchmarků není plně vyhovující pro naše měření. *Phoronix Test Suite* je nevyhovující z důvodu své přílišné rozsáhlosti a použití interpretovaného jazyka php. *Bonnie++* je příliš úzce zaměřený a nepodporuje všechny námi požadované testy, rozšiřování o další moduly není možné. Benchmarky *SPEC* nevyhovují svým zaměřením pouze na výkon procesoru a komerční licencí. *LMBench* se zabývá pouze měřením latence a propustnosti, chybí například všechny testy na měření výkonu procesoru. *LINPACK benchmark* se zabývá nejdůležitější oblastí z pohledu paralelního programování. Tento benchmark je nejblíže požadavkům naší práce, ale přesto není vhodný z důvodu absence testů některých důležitých částí, například síťového subsystému.

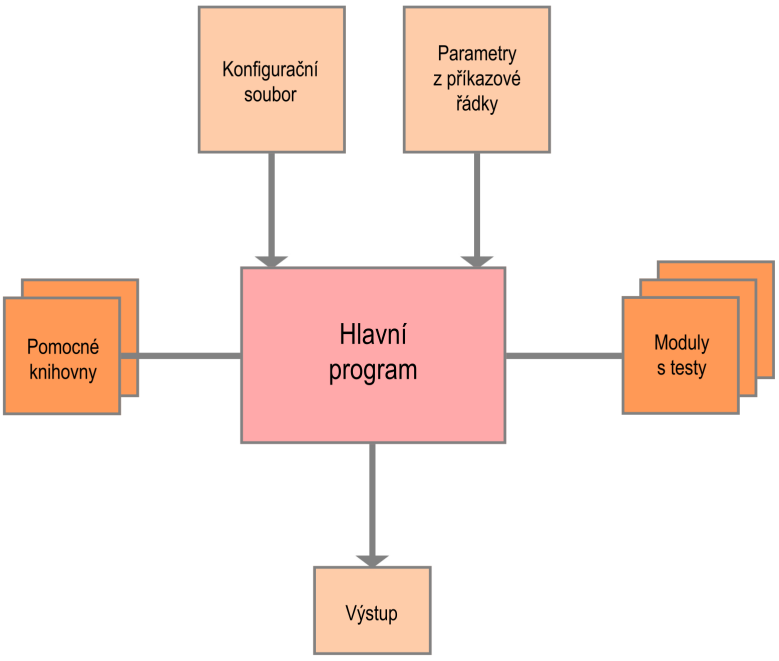
Protože žádný z představených benchmarků není plně vyhovující, rozhodli jsem se implementovat vlastní program s testy přesně podle našich požadavků. Jako programovací jazyk jsme vybrali C++. Omezení na podporu překladu do nativního kódu splňují ze známých a rozšířených jazyků Pascal, C a C++, přičemž Pascal se používá především pro výukové účely. Jazyky C a C++ jsou dnes standardem pro aplikace, po kterých je vyžadován vysoký výkon. C++ jsme zvolili kvůli jeho přednostem, které zahrnující pohodlnější vývoj programů včetně možnosti objektově orientovaného programování nebo lepší typovou kontrolu. Případné přejaté knihovny mohou být i v C. Vlastní implementace benchmarku také přinese plnou kontrolu nad celým systémem, což usnadní zpracování i interpretaci naměřených výsledků.

3. Návrh aplikace

V předchozí kapitole jsme odůvodnili potřebu vytvoření vlastního programu pro měření výkonu. Zásadní otázkou při tom je volba konkrétních testovacích algoritmů, čemuž jsme věnovali značnou pozornost. Všechny algoritmy jsme vybírali podle mnoha kritérií, mezi jinými i jednoduchost a přímočarost implementace, všeobecné rozšíření apod.

3.1 Hlavní program

Benchmark se skládá z několika základních funkčních bloků. Hlavním z nich je řídicí program, který zajišťuje zpracování parametrů z příkazové řádky, načtení a zpracování konfiguračního souboru, výpis výsledků, načítání a spouštění jednotlivých testů a měření doby jejich běhu. Další část tvoří kolekce všech testů seskupených do jednotlivých modulů dle cílové oblasti a typu testů. Všechny tyto moduly pak implementují předem dané rozhraní a jsou následně přeloženy jako dynamicky linkované knihovny, které jsou při spuštění řídicího programu automaticky načteny. Hlavní program může používat i některé další knihovny (například na zpracování konfiguračního souboru), které lze přilinkovat buď staticky nebo dynamicky. Základní schéma celého programu je na obr. 3.1.



Obrázek 3.1: Schéma architektury aplikace.

Tento návrh umožňuje pružně přidávat či upravovat jednotlivé moduly bez nutnosti zásahu do kódu samotného řídicího programu nebo nutnosti opětovné kompilace celého projektu.

Celý program spouští a vyhodnocuje jednotlivé testy, které byly implementovány v přidružených modulech. Jeho chování lze ovlivnit konfigurovatelnými hodnotami, které mohou být uvedeny v konfiguračním souboru nebo předány z příkazové řádky při spuštění. Lze upravovat parametry spouštěných testů i chování řídicí aplikace. Výstupem je ke každému testu reálný čas doby běhu a čas, po který testovaný kód programu aktivně běžel na procesoru. Tento výpis bude v pevném textovém formátu, který umožňuje snadné strojové zpracování.

3.2 Popis modulů

U každého modulu nyní uvedeme seznam testů, popíšeme použité algoritmy a oblasti, kterých se testování týká. Činnost jednotlivých testů lze do značné míry ovlivnit konfigurovatelnými parametry. Jejich výchozí hodnoty jsou zvoleny tak, aby na jedné straně respektovaly hardwarové možnosti Raspberry Pi (například alokace paměti bez nutnosti využití swapovacího oddílu) a zajišťovaly na něm trvání testu alespoň v řádu vteřin a na druhé straně aby doba běhu testu na nejrychlejším z testovaných počítačů nebyla výrazně nižší než desetina vteřiny. Tento časový rozsah byl určen jako rozumný kompromis mezi přesností měření a dobou běhu celého benchmarku. Nižší dolní limit pro dobu běhu testu by znamenal zvýšenou nepřesnost měření, protože by se do výsledné hodnoty mohly ve větší míře promítnout časy některých vnitřních procesů systému, například přepínání kontextu procesoru či velikost přidělovaných časových kvant jednotlivým úlohám.

3.2.1 Aplikační testy

Aplikační testy obsahují implementace známých a často využívaných algoritmů. Tím se snaží simulovat běžnou zátěž na systém jako celek, případně jen na některé jeho části. Tyto testy se nezabývají samostatně jednotlivými komponentami.

Kryptografické funkce

Modul **crypt** obsahuje šifrovací algoritmus AES [34] a dva algoritmy na výpočet hashovací funkce – SHA-256 [35] a Scrypt [36].

Kryptografické funkce mají v dnešní době poměrně široké využití. AES je pravděpodobně nejpoužívanější symetrická bloková šifra, která má malé paměťové nároky (okolo 4 kB) a její výpočet probíhá velmi rychle. Tento algoritmus dokáže ve velké míře využít rychlých pamětí cache, které na moderních procesorech mají již dostatečnou velikost. Některé procesory mají navíc i specializované instrukce urychlující výpočet. Test je spouštěn na konfigurovatelně velkém poli vyplněném náhodnými daty.

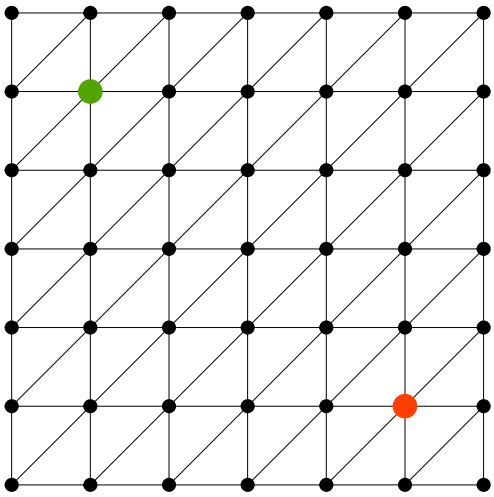
SHA-256 je kryptografická hashovací funkce s výstupem dlouhým 256 bitů. Hlavní používané operace jsou logické funkce, bitový posun a sčítání. Při testování se výsledný hash počítá ze zadaného textu, kterým je vyplněno vstupní pole určité velikosti.

Scrypt je hashovací funkce navržená pro ukládání otisků hesel či generování šifrovacích klíčů pro symetrické šifry. Algoritmus byl záměrně navržen jako vý-

početně pomalý převážně z důvodu vysoké paměťové náročnosti. Tento přístup má prakticky znesnadnit útoky hrubou silou.

Grafové algoritmy

V modulu **graph** nalezneme implementaci Dijkstrova algoritmu [37] pro hledání nejkratší cesty v kladně ohodnoceném grafu. Obsažená implementace používá k získání dosud nenavštíveného vrcholu s nejmenší vzdáleností od počátku 2-regulární haldu, tedy je vhodná především pro řídké grafy. Časová složitost algoritmu je pak $O((|V|+|E|)\log|V|)$. Struktura vstupního grafu je znázorněna na obr. 3.2, platí v něm $|E| = O(|V|)$ a jako každý rovinný graf jej lze považovat za řídký. Ohodnocení hran je generováno náhodně. Konfigurovat je možno velikost grafu, tj. počet vrcholů na straně čtvercové struktury. Počátek a konec hledané cesty jsou zvoleny v pevných pozicích vzhledem k protilehlým rohům čtvercové sítě (jsou to jediné dva vrcholy grafu, které mají právě čtyři sousedy stupně čtyři), viz obrázek.



Obrázek 3.2: Struktura vstupního grafu velikosti 7. Všechny hrany jsou orientovány oběma směry, počáteční vrchol je označen zeleně, cílový červeně.

Databázové algoritmy

V modulu **join** jsou implementovány dva databázové algoritmy na výpočet průniku dvou množin: hash join a merge join. Oba jsou implementovány ve dvou verzích: pouze 32bitové klíče a 32bitové klíče s 32bitovými hodnotami. Vstupní množiny jsou generovány náhodně. Hash join si nejprve vytvoří hashovací tabulku z menší množiny a poté pro každý prvek z druhé množiny hledá odpovídající klíč. Merge join nejprve obě množiny setřídí a poté při sekvenčním průchodu obou částí vybírá záznamy se stejným klíčem. Počty prvků obou množin lze dle potřeby upravovat.

Dynamické programování

Obsahem tohoto modulu je jeden test z oblasti dynamického programování, Wagner-Fischerův algoritmus [38] na výpočet Levenshteinovy editační vzdálenosti [39] dvou zadaných řetězců. V tomto testu je použit základní iterativní postup s plnou velikostí matice z důvodu vyšší paměťové náročnosti než má optimalizovaná verze algoritmu uchovávající vždy jen předchozí a aktuální řádek matice. Přístup do paměti je hlavní operací algoritmu, přičtení jedničky k celému číslu či nalezení minima ze tří čísel není výpočetně náročné.

Výpočty s reálnými čísly

Modul **matrix** obsahuje testy na rychlost zpracování operací s čísly s plovoucí desetinnou čárkou. Jsou použity dva algoritmy na násobení čtvercových matic řádu n, klasické násobení s časovou složitostí $O(n^3)$ a Strassenův algoritmus ($O(n^{\log_2 7})$) [40]. Ten je založen na celkovém snížení počtu násobení matic za cenu většího počtu jejich rychlejšího sčítání. Výhody tohoto algoritmu se projeví až u matic vyšších řádů.

Každý z testů je prováděn ve dvou verzích, které se liší použitými typy proměnných – `float` a `double`. Oba algoritmy jsou spouštěny na maticích se stejnými počátečními náhodnými daty. Jejich prvky jsou vhodně generovány tak, aby čísla ve výsledné matici zůstala mimo nestandardní hodnoty (například `+INF` či `NaN`).

Třídící algoritmy

Modul **sort** obsahuje jediný test, algoritmus Quicksort implementovaný v standardní knihovně jazyka C++ (`std::sort`). Jedná se o jeden z nejrozšířenějších a nejčastěji používaných třídících algoritmů teoreticky i prakticky, v průměrném případě má asymptotickou časovou složitost $\Theta(n \log n)$. Výhodou je třídění prvků bez další pomocné paměti přímo ve vstupním poli. Algoritmus je spuštěn na náhodně vygenerovaných datech (typu `int`).

Hlavní činnost tohoto algoritmu je porovnávání a prohazování položek v operační paměti. Porovnávání dvou celých čísel je na dnešních procesorech velmi rychlá operace, přístup do paměti je výrazně pomalejší, ale do jisté míry se zde také uplatní procesorové paměti cache.

Komprese dat

Komprese je hojně využívaná technika pro snížení velikosti informací. Z dostupných algoritmů byl vybrán DEFLATE [41], který je složením algoritmu LZ77 [42] s Huffmanovým kódováním [43] a svým použitím patří mezi nejrozšířenější. Použitá implementace pochází z open-source knihovny zlib [44].

Tato knihovna je velmi rozšířená, přenosná mezi různými platformami a mimo jiné také umožňuje konfiguraci velikosti používané operační paměti, takže ji lze úspěšně používat i na malých zařízeních s omezenými systémovými prostředky. Test probíhá tak, že vstupní text je nakopírován několikrát za sebe do připraveného pole, náhodně jsou změněna některá písmena a toto pole je komprimováno a následně dekomprimováno.

3.2.2 Syntetické testy

Jako syntetické označujeme ty testy, které měří parametry některé z komponent počítače, tedy jsou to spíše jednoúčelové a úzce zaměřené části programového kódu. Většina z nich by se dala zařadit do kategorie mikrobenchmark či I/O benchmark.

Paměťové testy

Modul **mem** testuje rychlost sekvenčního čtení a zápisu do operační paměti. Oba typy testů se provádí po několika různě velkých blocích, jako hodnoty pro zápis se používají čísla, jejichž binární reprezentace pravidelně střídá bity 0 a 1 (například `0x55` pro testy s 8 bitů velkým blokem). Tyto hodnoty jsou zvoleny kvůli doplňkovému testování vadných sektorů paměti (podobná čísla používají také běžné nástroje na testování paměti, např. *badblocks*). Test probíhá v konfigurovatelně velkém bloku paměti, jehož velikost by měla být přiměřená velikosti operační paměti Raspberry Pi.

Modul **cache** testuje vliv procesorových pamětí cache na rychlost čtení dat z operační paměti. Obsažen je jediný test, který se využije několika způsoby podle nastavení konfiguračních parametrů.

Při testu si vyhradíme souvislý blok paměti, z kterého opakovaně čteme čtyřbytové hodnoty na náhodně vybraných místech. V základním nastavení je paměťový blok dostatečně veliký, abychom při náhodném přístupu co nejčastěji požadovali data, která aktuálně nejsou uložena v paměti cache a musí se načíst přímo z operační paměti. Opakovaným spouštěním testu s různou velikostí pracovního bloku dat můžeme porovnat využití paměti cache a její velikost na všech testovaných počítačích. Začneme s malým blokem, který se celý načte do paměti cache, čímž prakticky eliminujeme přímé přístupy do operační paměti. Při každém dalším běhu testu velikost bloku zdvojnásobíme, takže od některého kroku bude narůstat pravděpodobnost toho, že požadovaná data bude nutné číst přímo z operační paměti, což se projeví zpomalením testu.

Test z modulu **cache** neměří přímo propustnost paměti při náhodném přístupu k datům. Ta se z operační paměti obvykle načítají po větších blocích, jejichž velikost je rovna velikosti základního bloku paměti cache (cache line). U procesoru BCM2835 se jedná o 32 bytů, desktopové i serverové procesory Intel moderních mikroarchitektur mají 64bytové cache line. Protože nám jde pouze o relativní porovnání počítačů při stejném testu, je tento návrh měření dostačující.

Testy persistentního úložiště

Rychlost čtecích a zapisujících operací s persistentním úložištěm, což je obvykle HDD, SSD nebo SD karta v případě jednodeskových počítačů, prověříme testy z modulu **card**. Test zápisu je pouze sekvenční, čtení je ve dvou verzích – sekvenční nebo s náhodným pořadím. Pro maximální rychlost je využíváno nízkoúrovňového API operačního systému místo odpovídajících prostředků standardní knihovny jazyka C++ (která navíc implementuje vyrovnávací paměť, chybovou diagnostiku apod.). Před spuštěním testů je vhodné vyprázdnit systémovou vyrovnávací paměť, aby čtení dat probíhalo přímo z testovaného úložiště a správnost výsledku nebyla negativně ovlivněna systémovými buffery.

Zápis testujeme tak, že připravené pole konfigurovatelné velikosti vyplníme náhodnými daty a opakovaně zapisujeme do souboru nastavené velikosti. Sekvenční čtení pak čte předpřipravený soubor opět po blocích stejné velikosti. U testu náhodného čtení lze samostatně konfigurovat velikost použitého bloku a počet iterací. Pozice ke čtení jsou generovány náhodně v odpovídajícím rozsahu k velikosti souboru.

Síťové testy

Modul **net** slouží k měření propustnosti a latence síťového subsystému. Pro správnou funkci je nutný kompatibilní síťový odpovídací program, který je spuštěn na přímo připojeném počítači podporujícím gigabitový Ethernet. Toto zapojení proti rychlejšímu počítači umožní měřit parametry pomalejšího Raspberry Pi. Odpovídací program naslouchá na nastavené adrese a portu a všechna příchozí data ihned odešle zpět. Testy na propustnost se provádí pro protokoly TCP i UDP, vždy po několika velikostech bloku, latence pouze přes UDP po malých blocích, aby nedocházelo k fragmentaci do více paketů. Všechna odesílaná data jsou náhodně generovaná.

4. Implementace a použití

V sekci 2.3 byly definovány naše požadavky na vhodný programovací jazyk a použitý operační systém. Pro implementaci benchmarku byl vybrán jazyk C++, přičemž přejaté zdrojové kódy jsou obvykle v jazyce C. K překladu je používána sada překladačů GCC ve verzi alespoň 4.8. Samotný překlad a linkování jsou řízeny utilitou GNU Make pomocí připraveného konfiguračního souboru `Makefile`. Cílový operační systém je GNU/Linux, proto budeme zachovávat běžné rozhraní programů pro tuto platformu, například syntax a sémantika přepínačů.

4.1 Popis hlavního programu

Celá aplikace se skládá z hlavního programu, který zajišťuje řízení celého testovacího procesu, a modulů s testy.

4.1.1 Načítání modulů

Jednotlivé moduly s testy se načítají dynamicky dle potřeby před začátkem testování. Všechny implementují stejné rozhraní, což zjednodušuje jejich použití i případné rozšiřování o další testy. Podrobnější popis rozhraní je v sekci 4.3.

Knihovny s moduly jsou umístěny ve stejném adresáři jako hlavní program. Jsou rozlišitelné pomocí prefixu `bm_` a přípony `.so` (např. `bm_sort.so`). V základním nastavení se použijí všechny nalezené knihovny, případně je možné jejich seznam upravit pomocí přepínačů. Moduly jsou pak dynamicky načítány a pomocí vytvořujících funkcí (factory functions) jsou získány instance sady testů v každém modulu. Každá sada obsahuje instance třídy reprezentující jednotlivé testy, jejichž zpracování je sekvenčně prováděno řídicím programem.

4.1.2 Průběh testování

Každý knihovní modul obsahuje jeden nebo více testů. Sada testů je reprezentována instancí třídy typu `test_suite` a každý jednotlivý test instancí třídy typu `test` (podrobněji viz 4.3).

Po zavedení knihovny se (dle nastavení přepínačů) načte seznam testů, které se postupně spouštějí. Pro každý test se provádí posloupnost následujících kroků:

1. Získání jména testu a jeho stručného popisu metodami `get_name()` a `get_info()`. Tyto údaje jsou použity při výpisu výsledku měření nebo obsažených testů.

2. Příprava prostředí pro test (inicializace proměnných, vytvoření testovacích dat apod.) vykonáním metody `prepare_environment()`. V případě výskytu chyby se testování ukončí a zpráva získaná zavoláním funkce `get_last_error()` je vypsána na standardní výstup.

3. Začátek měření času.

4. Spuštění testu zavoláním metody `run_test()`.

5. Konec měření času a vypsání času na výstup.

6. Kontrola správnosti výsledků zavoláním metody `check_result()`. I v případě záporného výsledku se pokračuje dále. Případná chybová zpráva (získaná `get_last_error()`) je vypsána na výstup.

7. Zavolání metody `evaluate(double time)`. Tento krok se provede jen v případě zapnutého rozšířeného výpisu a zároveň pokud kontrola výsledků proběhla úspěšně. V těle metody lze vypsat dodatečné informace k výsledku testu. Parametr s údajem doby běhu testu využije mimo jiné modul **mem** k vypsání rychlosti čtení dat z paměti.

8. Uvolnění všech použitých prostředků testu zajistí metoda `clear_environment()`.

4.1.3 Měření času

Časový interval, po který byl spuštěn nějaký program, lze měřit několika způsoby. Reálný čas reprezentuje celkovou dobu běhu algoritmu od jeho spuštění po úspěšné ukončení. Tento údaj je velmi důležitý, protože udává dobu, za kterou program od svého startu vydá výsledek. Procesorový čas udává součet časových úseků, které procesor aktivně strávil vykonáváním programu. Přesnost tohoto údaje je na některých počítačích včetně Raspberry Pi omezená, protože procesorové hodiny mají řádově nižší rozlišovací schopnost než reálné hodiny a výsledný procesorový čas je součtem více hodnot, čímž se naměřené nepřesnosti úměrně zvětšují. Do procesorového času se nezapočítává například pasivní čekání na dokončení probíhající vstupně-výstupní operace.

Měření času se provácí pomocí funkce `clock_gettime()`. Reálný čas měříme s parametrem `CLOCK_MONOTONIC`. Pro měření procesorového času si musíme nejprve zjistit identifikátor procesorových hodin, které následně použijeme jako parametr pro `clock_gettime()`. Počáteční i koncový čas se ukládají do struktury `timespec`. Ukázka kódu pro zjištění reálného a procesorového času:

```
timespec cpu_begin, real_begin;
clockid_t clockid;

clock_getcpuclockid(0, &clockid);
clock_gettime(CLOCK_MONOTONIC, &real_begin);
clock_gettime(clockid, &cpu_begin);
```

Popsaným způsobem lze na Raspberry Pi měřit procesorový čas s rozlišením na mikrosekundy a reálný čas s rozlišením na nanosekundy. Na běžném desktopovém počítači dosáhneme rozlišení obou těchto časů v nanosekundách. Alternativní přístupy pomocí funkcí `clock()`, `getrusage()` či `gettimeofday()` takové rozlišení nemají. Funkce `getrusage()` umožňuje rozlišit čas procesoru strávený v uživatelském prostoru od času stráveného v jádře, avšak přesnost na Raspberry Pi pouze na desítky milisekund je nedostatečná a neumožňuje její použití pro měření času v implementovaném benchmarku.

4.2 Konfigurace

Konfigurace programu je rozdělena na část týkající se hlavní aplikace a část pro moduly. Chování hlavního programu lze měnit pomocí parametrů předávaných z příkazové řádky při jeho spuštění. Tento způsob byl zvolen kvůli své jednoduchosti, snadnému a rychlému používání, nízkému počtu konfigurovatelných parametrů a zachování zvyklostí velké části unixových programů. Popis jednotlivých parametrů je uveden v uživatelské dokumentaci v sekci 4.6.1.

Prametrů pro jednotlivé moduly je velké množství, proto jsou uloženy v konfiguračním souboru. Z běžných textových formátů jako je JSON, INI nebo XML jsme zvolili JSON, protože umožňuje ukládat strukturovaná data v jednoduše čitelném formátu a existují knihovny pro jazyk C++ umožňující komfortní práci s těmito daty.

JSON (JavaScript Object Notation) [45] je jazykově nezávislý textový formát dat, který byl původně určen pro přenos údajů mezi servery a webovými aplikacemi. Data lze uchovávat v setříděných (pole) a nesetříděných (objekt) kolekcích. Pole mohou obsahovat hodnoty všech podporovaných typů – čísla, textové řetězce, logické hodnoty, pole, objekty a prázdnou hodnotu (`null`). Objekty obsahují dvojice klíč-hodnota, kde klíč je textový řetězec a hodnota libovolný podporovaný typ.

Pro práci s konfiguračním souborem byl vybrán parser `jsoncpp`[1], který je napsán v jazyce C++. O načtení konfiguračního souboru a jeho zpracování se stará hlavní program, který jednotlivým modulům předává odpovídající části konfigurace. Protože `jsoncpp` nepodporuje ověření struktury vstupního souboru pomocí JSON schématu[2], tak si správný typ svých parametrů musí hlídat každý modul zvlášť. Ve všech modulech existuje globální třída `config`, která načte potřebné parametry a uchová je pro pozdější použití.

4.3 Rozhraní modulů

Každý modul musí implementovat právě jednu třídu pro sadu testů (`test_suite`) a přinejmenším jeden testovací algoritmus. Všechny testy jsou potomky třídy `test` a implementují její čistě virtuální metody. Dále v modulu musí být obsažena vytvořující funkce `create()`, která vrátí instanci třídy reprezentující sadu testů. Následuje popis jednotlivých částí.

4.3.1 Třída test

```
class test {
public:
    virtual std::string get_name() = 0;
    virtual std::string get_info() = 0;
    virtual bool prepare_environment() = 0;
    virtual void run_test() = 0;
    virtual bool check_result() = 0;
    virtual void evaluate(double time) = 0;
    virtual std::string get_last_error() = 0;
```

[1] http://open-source-parsers.github.io/jsoncpp-docs/doxygen/index.html
[2] http://json-schema.org/latest/json-schema-validation.html

```
    virtual bool clear_environment() = 0;
    virtual ~test() {}
};
```

Každý test musí být opakovatelný, tj. všechna vstupní data si musí sám připravit a také je zodpovědný za alokaci a dealokaci paměti potřebné pro svůj výpočet. Průběh testování a pořadí volání jednotlivých metod z hlavního programu je popsán v sekci 4.1.2.

get_name(), get_info() Tyto metody vrací název a krátký popis testu, například „quicksort“, respektive „sorting vector of items“.

prepare_environment() Tato metoda má za úkol připravit data pro testování. Může se jednat o alokaci paměti, nastavení náhodného generátoru čísel, vyplnění pole daty, atd. Pokud je nastaven přepínač indikující rozšířený výpis, pak tato metoda vypíše konfigurovatelné parametry s jejich aktuálními hodnotami. Návratová hodnota indikuje, zda vše proběhlo úspěšně a je možno spustit testování.

run_test() V těle této metody je implementován vlastní algoritmus, jehož doba běhu bude měřena. Kontrola výsledků se provádí až později, aby nebylo ovlivněno měření času zpracování testu.

check_result() Jestliže je to z podstaty algoritmu v testu možné a snadno proveditelné, pak tato metoda zkontroluje správnost jeho výsledku. Korektnost výsledné hodnoty dává metoda vědět prostřednictvím návratové hodnoty. Pokud některý test nemá možnost rozumně ověřit výsledek, pak bude tato metoda vždy vracet `true`.

evaluate(double) Pokud test po skončení měřeného úseku získal nějaká výsledná data, která lze snadno vypsat na výstup, pak jsou zobrazena po zavolání této metody. K úpravě výsledku je možné použít vstupní parametr, který reprezentuje reálnou dobu běhu testu ve vteřinách, a zobrazit například rychlost práce algoritmu na jednotku dat.

get_last_error() Pokud nějaká metoda neuspěje, pak nastaví chybovou zprávu, která daný problém popíše. Tato metoda vrátí poslední takto nastavenou zprávu.

Za správné zpracování chybových stavů odpovídá hlavní program, kde je jejich obsluha pečlivě otestována a nemůže být ovlivněna přídavnými moduly. Tento přístup byl zvolen místo modernějších výjimek z důvodu možnosti lepší optimalizace výsledného kódu kompilátorem.

clear_environment() Tato metoda zajistí uvolnění všech použitých prostředků. To může znamenat dealokace paměti, uzavření souboru či socketu.

4.3.2 Třída test_suite

```
class test_suite {
public:
    std::vector<std::unique_ptr<test>> tests;
    void release();
    virtual std::string get_name() = 0;
    virtual std::string get_info() = 0;
    virtual void print_params() = 0;
    virtual ~test_suite() {}
};
```

Potomek této třídy musí být v celém modulu jedinečný. Stará se o kolekci všech testů v aktuálním modulu. Získání konkrétní instance se provede zavoláním vytvořující funkce v konkrétním modulu.

tests Seznam všech testů v aktuálním modulu.

release() Metoda k destrukci této třídy. Volána je z hlavního programu před ukončením modulu a zajišťuje správné uvolnění prostředků (vhodné provést v kódu modulu, kde proběhla alokace).

get_name(), get_info() Tyto metody vrátí jméno a krátký popis celého modulu, například „sort“, respektive „sorting data“.

print_params() Tato metoda vypíše konfigurační parametry celého modulu s jejich hodnotami.

4.3.3 Funkce pro vytvoření sady testů

```
typedef test_suite* create_t(bool, Json::Value&);
```

Vytvořující funkce v každém modulu musí dodržet tento typ a jméno „create“. Typická deklarace této funkce v každém modulu je:

```
extern "C" test_suite* create(bool verbose_flag, Json::Value&
    root);
```

Volající je odpovědný za korektní uvolnění veškerých prostředků pomocí metody `release()` po provedení všech testů.

4.4 Popis jednotlivých modulů

Následuje popis implementačně zajímavých částí každého modulu spolu s vysvětlením jednotlivých konfiguračních parametrů. U každé položky je vždy v závorce uveden typ a výchozí hodnota. Používají se konstanty dvou typů – *UInt* a *String*. V konfiguračním modulu má každý modul vlastní část, která je uvozena jeho jménem a uvnitř obsahuje všechny parametry daného modulu. Ukázka konfigurace modulu **graph**:

```
"graph" : {
    "side" : 512,
    "seed" : 42
}
```

Výchozí hodnoty parametrů byly experimentálně zvoleny tak, aby respektovaly hardwarová omezení Raspberry Pi a zároveň zajistily dostatečně dlouhou dobu běhu testu na všech testovaných počítačích.

Algoritmický návrh a základní popis testů je uveden v kapitole 3.

4.4.1 Modul cache

Tento modul testuje vliv procesorových pamětí cache na rychlost čtení dat z operační paměti. Obsahuje jediný test, jehož jméno ve výstupu programu je `cache`.

K rychlému generování náhodných indexů použijeme multiplikativní kongruenční metodu. Zvolíme $x_0 = 1$ a další čísla počítáme podle vzorce $x_{n+1} = p \cdot x_n \mod m$, kde x_n je předcházející vygenerované číslo, m je počet čtyřbytových čísel v bloku paměti a p je prvočíslo větší než m. Pak dostáváme pseudonáhodná čísla z intervalu $(0; m)$. Pro účely testu byla zvolena hodnota $p = 179425907$, což je prvočíslo větší než počet čtyřbytových hodnot v celé dostupné paměti RAM u Raspberry Pi.

Parametry

buffer_size	(*UInt*, 134 217 728) velikost bufferu v paměti (v bytech, tj. 128 MB)
iterations	(*UInt*, 13 421 772) počet prováděných čtení čtyřbytových hodnot

4.4.2 Modul card

Modul **card** testuje rychlost čtecích a zapisujících operací s persistentním úložištěm. Obsahuje tři testy, které se jmenují `seq_write`, `seq_read` a `rnd_read`.

Pro práci se soubory se z výkonnostních důvodů používají linuxové nízkoúrovňové funkce `open()`, `read()`, `write()`, `pread()`, `close()`, nikoli knihovní implementace funkcí pro vstup a výstup jako jsou například C++ streamy. V průběhu testování se vytváří dočasný soubor, který se po dokončení testu vždy odstraní.

Parametry

buffer_size	(*UInt*, 51 200) velikost bufferu (v bytech), který se používá u sekvenčních testů
repetition	(*UInt*, 1000) určuje kolikrát se buffer zapíše do souboru, respektive kolikrát je ze souboru načten. Velikost pomocného souboru (v bytech) je tedy $\textit{buffer\_size} \cdot \textit{repetition}$.
rand_reads	(*UInt*, 12 800 000) počet čtení v náhodném testu
rnd_read_buf_size	(*UInt*, 4) velikost bufferu (v bytech) při testu s náhodným čtením
seed	(*UInt*, 42) počáteční hodnota generátoru náhodných čísel pro zápis náhodných dat do souboru a pro pozici čtení v náhodném testu

file	(*String*, "./tmp_card_tests") cesta a jméno dočasného souboru používaného při testech v tomto modulu. Je třeba respektovat přístupová práva k souborovému systému.

Aby výsledky testů byly porovnatelné, je nutné, aby se vždy pracovalo se stejným objemem dat. Proto je podstatné zachovat rovnost

$$buffer\_size \cdot repetition = rand\_reads \cdot rnd\_read\_buf\_size.$$

4.4.3 Modul crypt

Tento modul obsahuje tři rozšířené kryptografické algoritmy. Jejich jména jsou `aes`, `sha256` a `scrypt`.

Ke všem testům v tomto modulu jsou použity referenční implementace algoritmů v jazyce C. Licenční ujednání je vždy součástí hlavního zdrojového souboru s implementací daného algoritmu.

Test u algoritmu AES provádí šifrování i dešifrování pro kontrolu správné činnosti, u ostatních dvou toto možné z principu není.

Parametry

aes_buf_size	(*UInt*, 8 388 608) velikost bufferu (v bytech) pro algoritmus AES
aes_seed	(*UInt*, 42) semínko generátoru náhodných čísel pro přípravu náhodných vstupních dat
sha256_text	(*String*, "The quick brown fox jumps over the lazy dog.") vstupní text pro výpočet SHA-256
sha256_buf_size	(*UInt*, 16 777 216) velikost pracovního bufferu algoritmu SHA
sha256_hash_size	(*UInt*, 32) délka výsledného hashe v bytech. Pro zachování algoritmu SHA-256 tuto hodnotu neupravujte (délka 64 bytů odpovídá algoritmu SHA-512).
scrypt_N	(*UInt*, 16 384) hodnota N pro algoritmus Scrypt
scrypt_r	(*UInt*, 8) hodnota r pro algoritmus Scrypt
scrypt_p	(*UInt*, 1) hodnota p pro algoritmus Scrypt
scrypt_key_size	(*UInt*, 64) velikost výsledného hashe algoritmu Scrypt (v bytech)
scrypt_text	(*String*, "pleaseletmein") vstupní text algoritmu Scrypt
scrypt_salt	(*String*, "SodiumChloride") pomocný text pro algoritmus Scrypt

Poznámka Parametry pro algoritmus Scrypt jsou podrobně vysvětleny v referenčním článku [36]. Složitost procesu lze ovlivnit hodnotami N, r a p. Tyto parametry ovlivňují počet prováděných iterací, paměťovou náročnost a náročnost na procesorový výkon. V této práci jsou použity doporučené hodnoty z referenčního článku.

4.4.4 Modul graph

Modul **graph** obsahuje implementaci Dijkstrova algoritmu pro hledání nejkratší cesty v grafu. Ve výstupu programu používá tento test jméno `dijk_fast`.

Hrany vstupního grafu pro Dijkstrův algoritmus jsou náhodně ohodnoceny celými čísly z intervalu $[1, 10]$. Graf je čtvercová síť s diagonálami napříč směru spojnice zdrojového a cílového vrcholu (viz obr. 3.2). Všechny hrany jsou orientovány oběma směry, ceny mohou být rozdílné.

Parametry

side	(*UInt*, 512) počet vrcholů na jedné straně čtvercové mříže, ve které se vytváří vstupní graf. Celkový počet vrcholů grafu je druhá mocnina této hodnoty.
seed	(*UInt*, 42) počáteční stav generátoru náhodných čísel pro přiřazování cen hranám grafu

4.4.5 Modul join

Databázové algoritmy hash join a merge join jsou implementovány ve dvou verzích v modulu **join**. Jména testů ve výstupu programu jsou `hash_1`, `hash_2`, `merge_1` a `merge_2`. Verze končící jedničkou pracují s množinami obsahujícími pouze klíče, verze končící dvojkou s množinami obsahujícími dvojice klíč – hodnota.

V algoritmu hash join je hashování obou množin prováděno prostředky jazyka C++, konkrétně použitím `std::unordered_set`.

Parametry

set1_count	(*UInt*, 1 300 000) velikost první vstupní množiny
set2_count	(*UInt*, 981 000) velikost druhé vstupní množiny
seed	(*UInt*, 42) počáteční stav generátoru náhodných čísel

4.4.6 Modul levenshtein

Tento modul obsahuje Wagner-Fischerův algoritmus dynamického programování na výpočet Levenshteinovy editační vzdálenosti dvou řetězců. Jméno testu ve výstupu programu je `levenshtein`. Z důvodu možnosti konfigurace vstupních řetězců není možné provádět kontrolu výsledků, proto lze výslednou hodnotu pro možnost kontroly vypsat.

Parametry

first	(*String*, ...) první vstupní řetězec
second	(*String*, ...) druhý vstupní řetězec

Poznámka Oba vstupní řetězce jsou dosti dlouhé, proto zde nejsou uvedeny výchozí hodnoty. Jejich délky jsou 3472 a 5269 znaků a vznikly zkopírováním náhodných odstavců anglicky psaného textu.

4.4.7 Modul matrix

Modul **matrix** testuje rychlost operací s čísly s plovoucí desetinnou čárkou. Obsahuje algoritmus pro klasické násobení matic a efektivnější Strassenův algoritmus. Jména testů ve výstupu programu jsou `multiply_4` a `strassen_4` pro typ `float` a `multiply_8` a `strassen_8` pro typ `double`.

Prvky vstupních matic jsou generovány náhodně s rovnoměrným rozdělením, které je symetrické podle středu $\frac{1}{\sqrt{n}}$ (kde n je velikost matice) a jehož levý okraj intervalu je desetina této hodnoty. To zajišťuje, že hodnoty ve výsledné matici budou dostatečně malé a nebude docházet k přetečení či podtečení datových typů v průběhu výpočtu. Pokud by se všechny prvky obou matic rovnaly střední hodnotě výše uvedeného rozdělení, pak by prvky výsledné matice měly hodnotu $n \cdot \frac{1}{\sqrt{n}} \cdot \frac{1}{\sqrt{n}} = 1$. Vstupní matice pro testy prováděné se stejným datovým typem jsou kvůli porovnání algoritmů identické.

Parametry

size	(*UInt*, 512) řád vstupních i výstupních matic (matice jsou čtvercové, tedy ve výchozím stavu 512×512)
seed	(*UInt*, 42) počáteční stav generátoru náhodných čísel

4.4.8 Modul mem

Testy paměti RAM (rychlost sekvenčního čtení a zápisu) probíhají v nastavitelných blocích. Čtení i zápis jsou prováděny ve verzích s velikostmi bloků 1 byte (`uint8_t`), 2 byty (`uint16_t`), 4 byty (`uint32_t`) a 8 bytů (`uint64_t`). Hodnoty, které se zapisují jsou vytvořeny z bytů `0x55` (`0x55`, `0x5555`, `0x55555555`, ...), aby se pravidelně střídaly bity 0 a 1. Po ukončení měřené části testu se provádí kontrola těchto hodnot v paměti. V případě jejího neúspěchu je podezření na přítomnost vadných bloků v operační paměti a doporučuje se provést další testování odpovídajícími programy.

Jména testů ve výstupu programu jsou `read_Xb` a `write_Xb`, kde `X` označuje velikost aktuálně používaného bloku.

Parametry

memory_size	(*UInt*, 134 217 728) velikost využité paměti v každém testu (v bytech, tj. 128 MB). Pro porovnání testů s různě velkým blokem je vhodné, aby všechny testy pracovaly s přesně stejným kusem paměti. Proto by velikost využité paměti měla být násobek velikosti největšího bloku, tj. 8 bytů.
read_loop	(*UInt*, 6) počet opakovaného čtení celého paměťového bloku ve všech čtecích testech
write_loop	(*UInt*, 6) počet opakovaného zápisu celého paměťového bloku ve všech zapisovacích testech

4.4.9 Modul net

Modul **net** slouží k měření propustnosti a latence síťového subsystému. Testy propustnosti probíhají po protokolech TCP a UDP s několika velikostmi přenášeného bloku. Tomu také odpovídají jména jednotlivých testů, která jsou `latency`, `tcp_2`, `udp_2`, `tcp_8`, `udp_8`, `tcp_32` a `udp_32`.

Všechny testy v tomto modulu probíhají na náhodných datech (generovaných `std::minstd_rand0`). Latence se měří pouze na UDP s malými bloky (které nebudou při přenosu fragmentovány), výsledek je průměrná doba odpovědi na paket. V testu nevyužíváme standardní ICMP datagramy jako u programu `ping`, protože pro práci s nimi je nutné administrátorské oprávnění a navíc mají přednostní zpracování v TCP/IP stacku, tudíž jejich použití pro měření parametrů síťového subsystému by mohlo dát zkreslené výsledky.

Ostatní testy probíhají po různě velkých blocích (2 KB, 8 KB a 32 KB), vždy po TCP i UDP. Celková velikost přenášených dat je jeden z konfigurovatelných parametrů. Výsledkem těchto testů je rychlost přenosu dat v Mbit/s (vypsaná při zapnutém rozšířeném výstupu). Samotné testy probíhají tak, že se odešle buffer nastavené velikosti přes síťový adaptér na pomocný počítač s běžícím odpovídačem `net-echo`. Ten obratem data pošle zpět a testovaný počítač data přijme. Tento postup se opakuje do vyčerpání vstupních dat. Po dokončení testu se provádí kontrola, zda přijatá data odpovídají odeslaným.

Předpokládá se, že počítač s odpovídačem je výrazně rychlejší než testovaný stroj a je připojen přímo. Za těchto podmínek je výsledek měření latence a propustnosti ovlivněn druhým zařízením jen ve velmi malé míře.

Parametry

test_size	(*UInt*, 8 388 608) celková velikost odesílaných dat v testech na propustnost (v bytech)
ping_count	(*UInt*, 4 096) počet odeslaných paketů při měření latence
ping_data	(*UInt*, 64) velikost bloku dat posílaných při měření latence (v bytech). Tato velikost by měla být dostatečně malá, aby nedocházelo k rozdělení dat do více paketů.
server	(*String*, "localhost") doménové jméno či IPv4 nebo IPv6 adresa vzdáleného počítače, proti kterému je prováděn test
tcp_port	(*UInt*, 10 000) TCP port na vzdáleném počítači
udp_port	(*UInt*, 10 000) UDP port na vzdáleném počítači
seed	(*UInt*, 42) počáteční stav generátoru náhodných čísel

Poznámka Raspbian má v továrním nastavení podporu IPv6 z důvodu úspory paměti vypnutou, lze ji však snadnou zapnout. Implementované testy podporují IPv4 i IPv6, ale v základním nastavení používají IPv4.

4.4.10 Modul sort

V tomto modulu je použita knihovní implementace třídícího algoritmu Quicksort, `std::sort`. Jako vstup dostává pole prvků typu `int`, které jsou náhodně generované s uniformním rozdělením. Jméno testu ve výstupu programu je `quicksort`.

Parametry

items	(*UInt*, 3 145 728) počet prvků ke třídění
seed	(*UInt*, 42) počáteční stav generátoru náhodných čísel

4.4.11 Modul zlib

Modul **zlib** obsahuje implementaci kompresního algoritmu DEFLATE z knihovny Zlib. Jméno testu ve výstupu programu je proto také `zlib`.

Zdrojové soubory komprimační knihovny jsou převzaty z referenční implementace v jazyce C. Původní text licence je součástí zdrojových souborů. Vstupní data se vytvoří opakováním konfigurovatelné znakové sekvence do zaplnění velikosti bufferu a náhodným přepisem některých písmen. Tento postup zajistí, aby algoritmus LZ77 nenahradil všechna opakování řetězce odkazem na první sekvenci, ale zároveň aby text mohl být i tímto algoritmem do jisté míry zkomprimován. Test měří čas komprese i dekomprese zároveň, kontroluje se shoda výsledku s originálními daty.

Parametry

buf_size	(*UInt*, 8 388 608) velikost vstupního bufferu pro komprimaci (v bytech)
text	(*String*, "The quick brown fox jumps over the lazy dog.") vstupní text, pomocí kterého je vyplněn buffer ke zpracování
seed	(*UInt*, 42) počáteční stav generátoru náhodných čísel

4.5 Adresářová struktura a sestavení

Zdrojové soubory se distribuují jako archiv `benchmark.tar.gz`. Po rozbalení se v aktuálním adresáři vytvoří složka `benchmark`, jejíž obsah bude mít následující strukturu.

- `config.json` Výchozí konfigurační soubor.
- `Makefile` Soubor, podle kterého se řídí vlastní překlad celé aplikace.
- `benchmark` Adresář hlavního programu.
  - `main.cpp` Zdrojový soubor hlavní aplikace.
- `bin` Adresář, do kterého se ukládají přeložené binární soubory. Pokud cílový adresář neexistuje, bude automaticky vytvořen při sestavení.
- `cache, card, ...` Adresáře obsahující zdrojové kódy pro jednotlivé moduly. Produktem překladu každého z nich je sdílená knihovna, jejíž jméno vznikne doplněním prefixu `bm_` a přípony `.so`, například `bm_cache.so`, `bm_card.so`.
  - `main.cpp` Hlavní zdrojový soubor. Podle potřeby obsahují některé adresáře i další zdrojové soubory.

- `includes` Adresář pro sdílené hlavičkové soubory.
  - `test.h` Deklarace třídy testu, sady testů a typu vytvořující funkce.
  - `json.h` Hlavičkový soubor pro parser `jsoncpp`.
- `net-echo` Adresář pomocného programu pro síťové testy.
  - `net-echo.cpp` Zdrojový soubor pomocného programu na měření síťových testů.
  - `Makefile` Soubor, podle kterého se řídí překlad pomocného programu `net-echo`.
- `jsoncpp_lib` Adresář pro zdrojové soubory parseru `jsoncpp`, který je překládán jako sdílená knihovna. Výsledný soubor má název `libjsoncpp.so` a je za překladu přilinkován k hlavnímu programu.
  - `main.cpp` Sloučený zdrojový kód parseru podle skriptu `amalgamate.py` z repozitáře projektu[3]. Odpovídající hlavičkový soubor je přesunut do složky `includes` místo původního umístění v podsložce `json`.

Překlad aplikace včetně všech částí se řídí připraveným souborem `Makefile`. Podporovány jsou tři základní cíle – výchozí *all*, *clean* a *purge*. Cíl *all* znamená překlad souborů, jejichž zdrojový kód se od posledního překladu změnil, nebo všech souborů, ke kterým nejsou dostupné jejich objektové ekvivalenty (soubory s příponou `.o` v podadresářích složky `bin`). Cíl *clean* smaže tyto objektové soubory a zachová přeložený hlavní program spolu se všemi moduly, kdežto cíl *purge* provede smazání všech souborů vzniklých při překladu.

Vzhledem ke skutečnosti, že program je na konkrétním počítači určen spíše k jednorázovému použití (nasbírání potřebných dat), není prováděna jeho instalace a začlenění do operačního systému, tj. nejsou podporovány cíle *install* (instalace knihoven, manuálové stránky, atd.) a doplňkový *uninstall*.

Pomocný program pro síťové testy se překládá pomocí vlastního souboru `Makefile` umístěného ve složce `net-echo`, kde je poté vytvořen i výsledný binární spustitelný soubor. Jediný podporovaný cíl překladu je *all*.

4.6 Uživatelská dokumentace

Aplikace se spouští z příkazového řádku. Její chování závisí pouze na nastavení jednotlivých přepínačů, po spuštění již nelze průběh jakkoli ovlivnit.

4.6.1 Přepínače

Zpracování přepínačů předaných z příkazové řádky při spuštění se provádí pomocí knihovní funkce `getopt_long()` deklarované v hlavičkovém souboru `getopt.h`. To zaručuje rozpoznávání jak krátkých (`-h`), tak i dlouhých verzí přepínačů (`--help`) a také slučování krátkých verzí (například místo `-l -v` lze napsat `-lv`). Podporované přepínače jsou tyto:

[3] `https://github.com/open-source-parsers/jsoncpp`

```
benchmark [-h][-a | -i list | -e list][-v][-l][-t number][-c file]
```

`-h` (`--help`) vytiskne seznam všech dostupných přepínačů a krátkou nápovědu ke každému z nich

Následující trojice přepínačů ovlivňuje to, které moduly budou načteny. Lze použít nejvýše jeden z nich. Pokud nebude uveden žádný, chování aplikace bude odpovídat přepínači `-a`.

- `-a` (`--all`) spustí všechny testy, které najde ve stejném adresáři s hlavním programem a jejichž název odpovídá regulárnímu výrazu `bm_[^\.]+\.so`.
- `-i` (`--include`) spustí pouze vyjmenované testy. Přepínač má povinný argument, který obsahuje seznam jmen modulů oddělených čárkou. Například `-i mem,net,crypt`
- `-e` (`--exclude`) spustí všechny testy kromě těch, které jsou vyjmenovány. Přepínač má povinný argument, který obsahuje seznam modulů oddělených čárkou (stejné jako u `-i`).

Ostatní podporované přepínače jsou uvedeny níže.

- `-v` (`--verbose`) zapne podrobnější výpis. Lze použít při spuštění testů nebo při výpisu informací o modulech přepínačem `-l`.
- `-l` (`--list`) vypíše seznam načtených modulů. S přepínačem `-v` vypíše navíc ke každému modulu seznam obsahujících testů a konfigurační parametry náležejcí k danému modulu.
- `-t` (`--times`) určuje, kolikrát se má každý test spustit. Přepínač má povinný argument, který obsahuje celé kladné číslo.
- `-c` (`--config`) nastaví cestu a název konfiguračního souboru. Argument přepínače je povinný, může obsahovat relativní i absolutní cestu, případně pouze jméno souboru, pokud je ve stejném adresáři s hlavním programem. Pokud přepínač není uveden, program zkusí najít soubor `config.json` v adresáři hlavního programu. V případě jeho neexistence se použijí ve všech případech výchozí hodnoty.

4.6.2 Výstup

Veškeré informace program vypisuje na standartní výstup. Je tedy možné využít různých možností příkazového řádku jako například přesměrování standardního výstupu do souboru.

Při spuštění programu je na každé řádce vypsáno jméno testu a dva časy, vždy oddělené právě jednou mezerou. První čas je reálný čas doby běhu bez ohledu na plánování vlákna na procesoru a dalších okolnostech. Druhý čas je čas skutečně využitý procesorem (CPU time). Tento čas charakterizuje využití samotného procesoru (tj. není započítána režie operačního systému, čekání na periferie apod.). Oba časy jsou uváděny v sekundách jako desetinná čísla.

Při zapnutí rozšířeného výpisu pomocí přepínače `-v` jsou informace podrobnější, zaznamenává jednotlivé kroky, konfigurační parametry, případně vyhodnocuje výsledek měření.

Vzorový výstup pro několik kombinací přepínačů hlavního programu i pro pomocný program `net-echo` je v příloze A.

4.6.3 Návratové hodnoty

Pokud program neočekávaně skončí, vypíše na výstup důvod svého ukončení, případně lokalizaci chyby. Návratové hodnoty programu, například pro ošetření chyb ve skriptech, jsou následující:

- **0** žádná chyba
- **1** spouštěcí parametry neodpovídají specifikaci
- **2** nepovedlo se načíst moduly testů
- **3** nepovedlo se načíst parametry z konfiguračního souboru (pravděpodobná příčina je syntaktická chyba v souboru)
- **4** některý parametr v konfiguračním souboru má chybný typ

4.6.4 Program net-echo

```
net-echo [-h][-t port][-u port]
```

Pomocný program `net-echo` se spouští podobným způsobem jako hlavní aplikace. Podporuje tři přepínače – `-h` (`--help`), `-t` (`--tcp`) a `-u` (`--udp`). První zmíněný vypíše krátkou nápovědu, další dva přijímají jeden povinný parametr a to číslo příslušného portu, na kterém probíhá síťová komunikace. Pokud nejsou uvedeny, výchozí hodnota obou přepínačů je 10 000.

Zde bychom si dovolili připomenout, že bez administrátorských oprávnění nelze používat porty v rozsahu do 1024. Pro úspěšné navázání komunikace může být nutné na obou propojených počítačích upravit pravidla firewallu.

5. Měření a zpracování dat

Pro měření a zhodnocení použitelnosti jednodeskového počítače Raspberry Pi jako jednotky výpočetního clusteru jsme použili dvě verze tohoto počítače, model B a vylepšený model B+. Testovaný model B je vybaven SDHC kartou značky PRETEC s kapacitou 32 GB a udávanými rychlostmi čtení a zápisu až 60 MB/s, respektive 35 MB/s. Model B+ byl testován s microSDHC kartou Kingston, jejíž kapacita podle specifikace je 32 GB, rychlost čtení až 90 MB/s a rychlost zápisu až 45 MB/s.

Pro porovnání výkonu Raspberry Pi s desktopovými a serverovými počítači jsme zvolili z obou kategorií jedno referenční zařízení. Vybraný desktopový počítač je osazen čtyřjádrovým procesorem Intel Core i7 870 mikroarchitektury Nehalem s taktovací frekvencí 2,93 GHz. Dále obsahuje 16 GB paměti RAM (typ DDR3-1600) a dvojici 100 GB SSD disků značky Kingston zapojených do RAID 1 (zrcadlení disků). Jako operační systém je použit Red Hat Enterprise Linux 7.

Vybraný server je Dell PowerEdge M910. Jedná se o NUMA (Non-Uniform Memory Access) počítač se čtyřmi osmijádrovými procesory Intel Xeon E7 4820 mikroarchitektury Nehalem s taktovací frekvencí 2 GHz doplněnými 32 GB paměti RAM (tj. 128 GB celkem). K serveru je připojeno diskové pole Infortrend ESDS 3060, které obsahuje dvojici 400 GB SSD disků sloužících jako vyrovnávací paměť a 14 kusů 4 TB HDD disků s 7200 otáčkami za minutu. Operační systém je (stejné jako v případě referenčního desktopového počítače) Red Hat Enterprise Linux verze 7.

5.1 Spotřeba energie

Výrobcem udávaná spotřeba Raspberry Pi je okolo 3,5 W, respektive 2,5 W u novějšího modelu B+. Její hodnoty závisí i na množství připojených periferií, proto ji budeme ověřovat v konfiguracích podobných podmínkám jednotlivých zařízení uvnitř výpočetního clusteru – a aktivním síťovým připojením, bez monitoru, klávesnice a jiných periferií.

Příkon[1] Raspberry Pi budeme měřit na napájecím kabelu mezi zdrojem a počítačem. Běžně rozšířené přístroje na měření spotřeby elektrické energie nemají dostatečnou přesnost pro rozlišení malých odběrů, proto zde provádíme měření hodnot stejnosměrného napětí a proudu a výsledný příkon spočítáme podle vzorce $P = U \cdot I$. Použitý měřící přístroj je KCX-017, který dle specifikace výrobce měří stejnosměrné napětí v rozsahu 3 až 7 V s přesností ± 0,04 V a proud v rozmezí 0,05 až 3,5 A s přesností ± 0,02 A. Účinnost nápájecích zdrojů použité velikosti se pohybuje okolo 80 %, proto tuto hodnotu v celkovém příkonu také zohledníme.

Spotřeba Raspberry Pi verze B se podle zatížení pohybovala mezi 2,3 až 2,8 W (po startu při odpojené síti pouze 2,1 W). Kromě síťového připojení měla největší vliv práce s paměťovou kartou.

U modifikovaného modelu B+ i naše měření potvrdilo významnou úsporu energie. Po startu bez připojených periferií dosahovala spotřeba 1,2 W, jako nejvyšší hodnotu jsme zaznamenali 1,7 W (při běhu modulu **crypt**).

[1] Příkon udává množství energie spotřebované za jednotku času.

Ke každému uzlu navíc připočítáme spotřebu 0,5 W, která pokrývá odhadovanou spotřebu síťových prvků zajišťujících propojení uvnitř clusteru. V následujících úvahách budeme používat hodnotu 3,3 W pro starší model B a 2,2 W pro úspornější model B+. V obou případech je již započtena účinnost napájecího zdroje.

Spotřebu desktopového počítače kvalifkovaně odhadneme pomocí dostupných informací o podobných typech. Účelem tohoto testování je provést obecné srovnání spotřeby clusteru s desktopovým počítačem, proto nám nezáleží na přesné hodnotě příkonu konkrétního kusu. Přímé měření by neposkytlo relevantní údaje, protože spotřebu výrazně ovlivňuje další připojený hardware, zejména dvojice výkonných grafických karet, které jsou z pohledu našeho testování nevýznamné. Výsledná uvažovaná hodnota příkonu pro další zpracování je 250 W.

U serverového počítače zjistíme spotřebu podle údajů šasi (blade chassis), kde je zapojený. K výsledku musíme také amortizovat odhad spotřeby podpůrného hardware, hlavně šasi, které zajišťuje například síťové připojení nebo napájení jednotlivých blade serverů. Jedno šasi většinou obsahuje více těchto jednotek, proto je možné jeho vlastní spotřebu rozpočítat mezi jednotlivé servery. Dle údajů na indikátoru počítačové skříně se příkon jednoho serveru pohyboval okolo 275 W v klidu, se vzrůstající zátěží rostl až na 350 W. Po započtení vlivu dalšího podpůrného hardwaru budeme dále pracovat s hodnotou 500 W.

Spotřebu síťových prvků, diskového pole a chlazení zanedbáváme, protože tyto součásti bychom potřebovali u clusteru z Raspberry Pi i u blade serveru, tudíž na relativní porovnání nemají významný vliv.

5.2 Měření výkonu

Hlavním cílem této práce je odhadnout výpočetní schopnosti clusteru složeného z Raspberry Pi a porovnat je s běžnými počítači. Všechna měření byla provedena pomocí jednovláknových aplikací. Dále budeme vycházet z představy dokonale paralelizovatelné úlohy, která plně využívá všech dostupných fyzických jader systému[2] a u níž nedochází k omezování výpočtu jednotlivých vláken způsobeným například vzájemnou synchronizací při přístupu ke sdíleným datům. K takovému idealizovanému systému budeme extrapolovat výsledky našich měření.

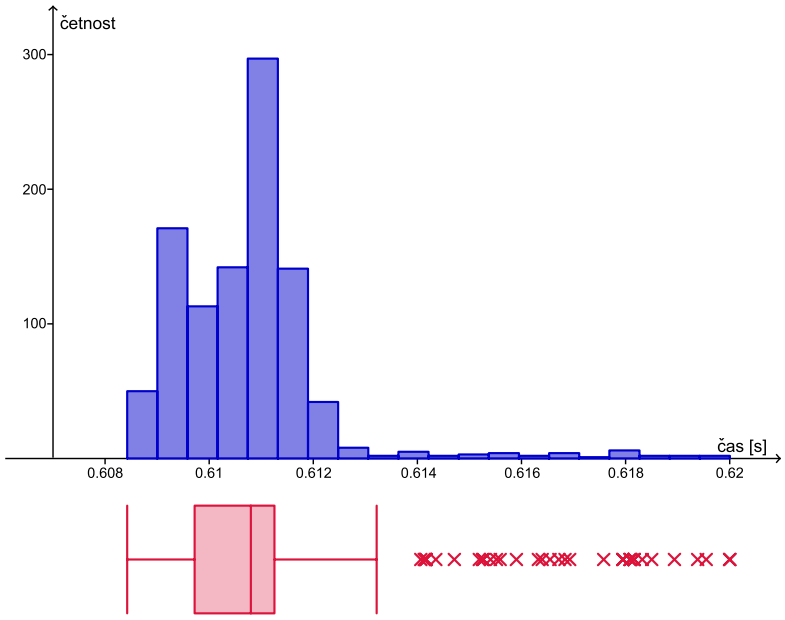
5.2.1 Metodika

Výpočetní výkon budeme měřit pomocí připraveného benchmarku, který bude spuštěn na všech testovaných počítačích. Abychom mohli následně provést porovnání výsledků, budeme používat vždy stejné konfigurační parametry. S ohledem na omezené možnosti jednodeskového počítače Raspberry Pi jsme zvolili výchozí hodnoty konfigurovatelných parametrů, které byly přímo připraveny pro plné využití tohoto malého počítače bez vedlejších jevů jako je například přesun paměťových stránek mezi operační pamětí a diskem.

[2]U procesorů Intel přitom odhlédneme od skutečnosti, že díky technologii Hyper-Threading se každé fyzické jádro jeví jako dvě jádra logická. Reálné zvýšení výkonu je zdrojem řady polemik. Optimistické odhady uvádějí v závislosti na druhu aplikace vzrůst o 15 až 30 %, řada kritiků však poukazuje i na množství problémů, např. `http://en.wikipedia.org/wiki/Hyper-threading`.

Čas běhu testu je částečně ovlivněn i prostředím, ve kterém je spouštěn. Proto při opakování testu se stejnými daty dostaneme mírně odlišné výsledky. Část změřené doby je využita na zpracování jiných úloh (systémových, jako například zpracování asynchronních operací spojených s obsluhou hardware, nebo uživatelských, tj. ostatních programů spuštěných uživateli systému) a přepínání kontextu mezi nimi, což ovlivňuje také obsazení procesorových cache i dalších vyrovnávacích pamětí operačního systému a tím nepřímo i čas testu.

Pro posouzení těchto (z našeho hlediska náhodných) vlivů jsme jeden vybraný test (třídění pole metodou quicksort) nechali proběhnout 1000 krát a zaznamenávali naměřené hodnoty. Histogram na obrázku 5.1 reprezentuje rozdělení naměřených časů (interval mezi minimální a maximální hodnotou je rozčleněn na 20 dílů). Můžeme konstatovat, že rozptyl hodnot je poměrně malý a rozdělení je výrazně asymetrické. Směrem ke kratším časům je ostře ohraničeno, pro delší časy jsou četnosti na významném intervalu nízké, ale nenulové. Dokládá to i tzv. krabicový diagram (boxplot), na obrázku pod časovou osou. Vybarvený obdélník má hranice v dolním a horním kvartilu, čára uvnitř znázorňuje medián všech hodnot. Variabilita dat je vyznačena tzv. vousy (whiskers – svislé čáry znázorňující minimum a maximum po vyřazení odlehlých hodnot), body ležící mimo jsou vykresleny jednotlivě a jsou pokládány za odlehlé hodnoty (outliers – 1,5 násobek mezikvartilové vzdálenosti od přilehlého kvartilu). Stejné vyhodnocení hodnot procesorového času ukazuje, že rozdělení je kompaktnější (minimum zůstává prakticky stejné, výrazně však klesá množství odlehlých hodnot směrem k vyšším časům), což potvrzuje naše předpoklady.



Obrázek 5.1: Statistické rozdělení naměřených hodnot třídícího tesu.

Na základě provedené analýzy budeme časy testů zpracovávat takto:

- Každý test spustíme 20 krát s identickou konfigurací. To nám zajistí dostatečně veliký soubor výsledných časů pro spolehlivé odfiltrování odlehlých hodnot a zároveň celé testovaní bude trvat přiměřeně dlouhou dobu.
- Vyřadíme odlehlé hodnoty pro reálný i procesorový čas odděleně. Hodnotu prohlásíme za odlehlou, pokud je menší než $q_{25} - 1,5 \cdot (q_{75} - q_{25})$ nebo větší než $q_{75} + 1,5 \cdot (q_{75} - q_{25})$, kde q_{25} je dolní a q_{75} horní kvartil statistického souboru.
- Ze zbylých hodnot spočteme aritmetický průměr, který v dalším porovnání použijeme jako reprezentativní hodnotu trvání testu. Pro případnou detailnější analýzu uvádíme u výsledných hodnot též směrodatnou odchylku, kterou lze použít k různým doplňkovým zpracování naměřených dat mimo rozsah této práce.

Síťové testy provádíme pouze u Raspberry Pi, protože ve výpočetním clusteru budou tato zařízení vzájemně propojena pomocí Ethernetu. Z tohoto důvodu mají na celkový výpočetní výkon clusteru určitý vliv latence a propustnost síťového subsystému. Srovnávaný desktopový a serverový počítač mají řádově rychlejší síťovou techniku a jejich komunikace je méně intenzivní, proto parametry síťového subsystému přímo nesouvisí s výpočetním výkonem zařízení.

Uvedeným postupem jsme na všech čtyřech srovnávaných počítačích provedli opakované měření času běhu testů. Souhrnné tabulky se statisticky zpracovávanými daty jsou uvedeny v příloze B.

Celkově lze konstatovat, že relativní chyba naměřených časů je malá (typicky < 1 %), výjimkou jsou zvláště některé testy pro práci s persistentním úložištěm, kde má vliv i momentální stav souborového systému. Kromě úloh, kde se pracuje s periferiemi (persistentní úložiště, síťový subsystém) je celkový čas prakticky shodný s procesorovým časem, proto je jeho využití v následujícím zpracování opodstatněné.

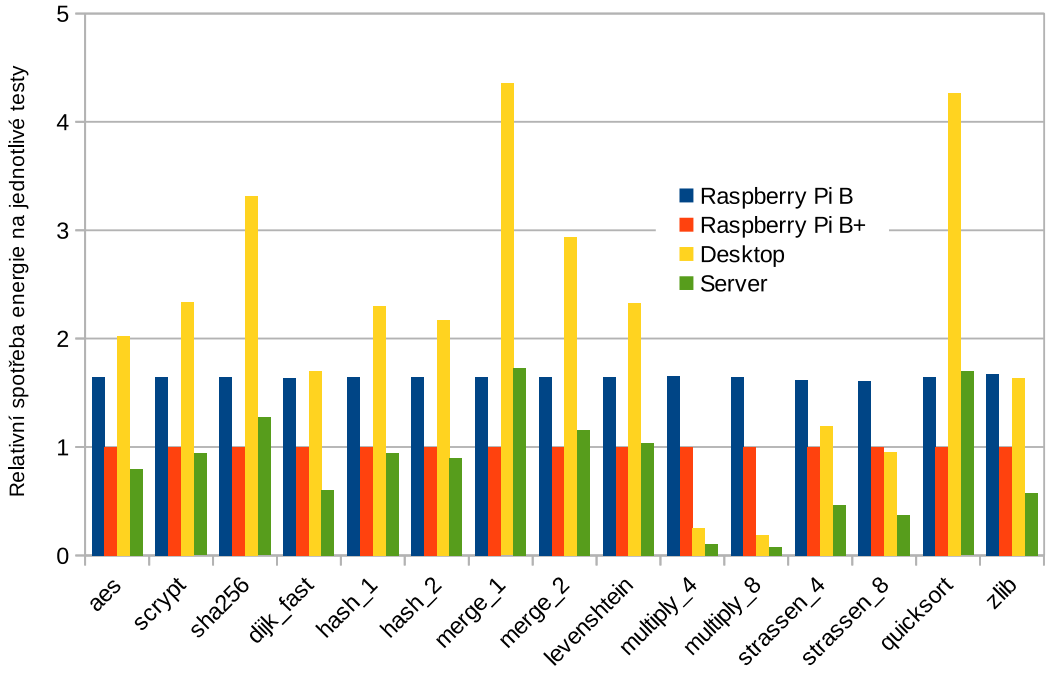
Časy pro Raspberry Pi B a B+ můžeme v rámci statistických chyb považovat za stejné, větší odchylku u zápisu na paměťovou kartu přisuzujeme jejich hardwarové odlišnosti. Tím se při našem měření potvrdilo, že z výpočetního hlediska se jedná o dvě totožná zařízení.

5.2.2 Aplikační testy

Aplikačními testy jsme porovnali výkon počítačů při výpočtu několika vybraných běžných a často používaných algoritmů. Pro připomenutí uvedeme jejich stručný přehled spolu s hlavními konfiguračními parametry. Mezi implementované kryptografické algoritmy patří AES, který je spuštěn na datech o velikosti 8 MB, SHA-256, který počítá hash z dat dvojnásobné velikosti proti AES a Scrypt, jehož konfigurace je převzata z referenčního článku. Grafové algoritmy zastupuje Dijkstrův algoritmus na výpočet nejkratší cesty mezi dvěma vrcholy v ohodnoceném grafu, jehož vstup je rovinný graf s vrcholy uspořádanými do čtvercové mříže s hranou 512 bodů. Databázové algoritmy hash join a merge join spojují dvě množiny s velikostí okolo milionu prvků. Editační vzdálenost je počítána

Wagner-Fischerovým algoritmem z řetězců o délce přibližně 3500 a 5300 znaků. Násobení reálných matic klasickým i Strassenovým algoritmem probíhá na čtvercových maticích řádu 512. Pro třídění čísel je použita vstupní množina se zhruba 3 miliony prvků. Komprimační knihovna provádí výpočet na datech o velikosti 8 MB.

Výsledky měření jsou znázorněny na obrázku 5.2. Zdrojová data reflektují energetickou náročnost dané úlohy a byla získána jako součin doby běhu testu a příkonu vztaženého na jedno fyzické jádro. Tato veličina představuje energii vynaloženou na provedení výpočtu. Hodnoty jsou vyneseny v relativním měřítku se vztažnou jednotkou Raspberry Pi model B+. Díky tomu můžeme výsledky všech aplikačních testů zobrazit v jednom grafu s možností snadného porovnání. Relativní měřítko je použito ke kompenzaci často výrazně odlišných energetických náročností jednotlivých testů, ale se zachováním poměrů mezi jednotlivými počítači u každého testu.



Obrázek 5.2: Relativní porovnání energetické náročnosti aplikačních testů.

Rozdíly mezi modely Raspberry Pi pramení prakticky pouze z hodnoty příkonu. Mezi procesory Intel vychází v tomto srovnání lépe serverový Xeon, spotřeba energie byla obecně méně než poloviční v porovnání s Core i7. Energetická náročnost na provedení úlohy u ARM v Raspberry Pi je většinou nižší než u Core i7, u některých testů dosti výrazně. Celkově lze říci, že úlohy náročné na paměť RAM vychází lépe na Raspberry Pi a naopak procesory Intel jsou výhodnější pro úlohy potřebující hrubý výpočetní výkon (například násobení matic).

Jedinou výraznou odchylkou mezi výsledky testů jsou výpočty v plovoucí desetinné čárce. Maticové násobení klasickým algoritmem je i na desktopu velmi výrazně úspornější než u Raspberry Pi. Použití Strassenova algoritmu se stejnými maticemi vykazuje daleko menší rozdíly mezi testovanými počítači. Změna doby běhu testu mezi klasickým a Strassenovým algoritmem u procesorů Intel není příliš významná, naproti tomu u ARM pozorujeme více než pětinásobné zrychlení.

Díky FPU moderních procesorů jsou rozdíly mezi sčítáním a násobením čísel s plovoucí desetinnou čárkou malé nebo žádné[3]), proto zjištěné zrychlení přičítáme skutečnosti, že ve Strassenově algoritmu se úloha rekurzivně dělí na menší podúlohy, takže od jistého kroku se veškerá data vejdou do paměti cache. Použité matice řádu 512 zabírají v paměti $512^2 \cdot 4$ B = 1 MB respektive dvojnásobek pro typ `double`, takže se obě zdrojové i výsledná matice najednou vejdou do L3 cache testovaných procesorů Intel. Proto se vliv těchto pamětí projeví i u klasického algoritmu, zatímco cache procesoru ARM lze využít až při úlohách menšího řádu ve Strassenově algoritmu.

5.2.3 Doplňující syntetické testy

Paměť

Rychlost přístupu do paměti (čtení instrukcí programu a zápisu i čtení dat) je faktor, který ovlivňuje dobu běhu každého programu. Testování propustnosti operační paměti při sekvenčním zápisu a čtení dat provádí testy z modulu **mem**. V tabulce 5.1 jsou uvedeny naměřené hodnoty pro čtyři testované počítače při různě velkých blocích dat.

	Raspberry Pi B	**Raspberry Pi B+**	**Desktop**	**Server**
read_4b	124,3	117,4	5962,3	2555,1
read_8b	128,5	128,5	10562,0	2194,1
write_4b	328,8	328,8	5969,9	2622,1
write_8b	596,2	596,0	5830,8	2563,2

Tabulka 5.1: Operační paměť – přenosové rychlosti v MB/s při čtení a zápisu dat po 32bitových a 64bitových blocích.

Obecně můžeme konstatovat, že při větších blocích dat dosahujeme vyšší přenosové rychlosti. Rozdíl mezi testy s 4 bytovou a 8 bytovou hodnotou je v počtu prováděných strojových instrukcí, protože přístup do paměti probíhá v obou případech po blocích, jejichž velikost odpovídá velikosti cache line[4]. Výrazný nárůst rychlosti některých testů s 8 bytovým blokem proto přisuzujeme právě tomuto rozdílu. Naopak zpomalení při testu čtení většího z bloků na serveru lze vysvětlit NUMA architekturou počítače. Pro podrobnější závěry by bylo potřeba provést dodatečná testování.

Vyrovnávací paměť

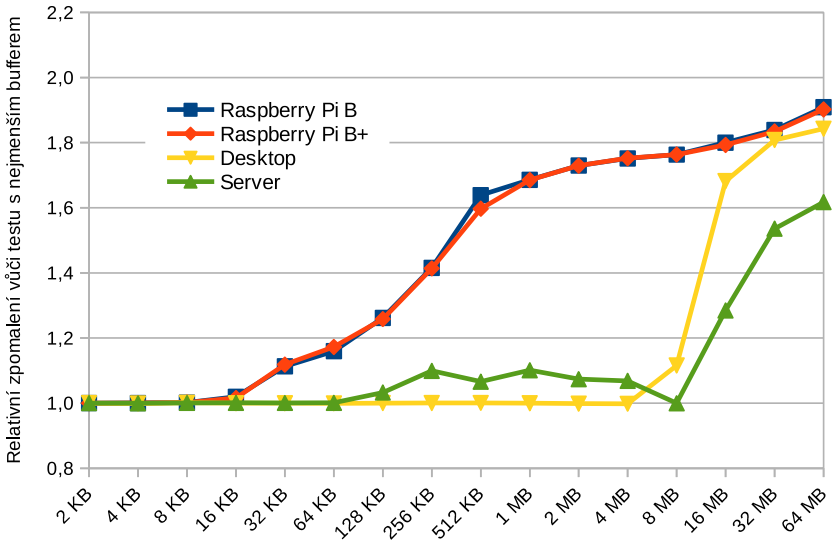
Významnou součástí architektury moderních procesorů, která výrazně ovlivňuje jejich výkon, jsou asociativní vyrovnávací paměti (cache). Oba použité procesory Intel mají tříúrovňovou datovou paměť cache, L1 má kapacitu 32 KB a L2 má 256 KB (pro každé jádro), L3 je společná pro všechna jádra. Její kapacita je

[3]Instrukce FADD (sčítání) a FMUL (násobení) u mikroarchitektury Intel Nehalem viz `http://www.agner.org/optimize/instruction_tables.pdf`, str. 150, obdobné instrukce pro FPU jednotku procesoru ARM viz `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0301h/DDI0301H_arm1176jzfs_r0p7_trm.pdf`, kapitola 21.11.

[4]základní kopírovatelná jednotka mezi hierarchickými úrovněmi pamětí

8 MB u Core i7 (desktop), respektive 18 MB u Xeon (server). ARM procesor v Raspberry Pi má 16 KB cache první úrovně, L2 má kapacitu 128 KB[5].

Pro posouzení vlivu cache jsme nechali proběhnout test z modulu **cache** s různými parametry. Postupně jsme měnili velikost zdrojového bufferu v rozsahu 2 KB až 64 MB a sledovali vliv na dobu běhu testu. Výsledky shrnuje obr. 5.3, hodnoty znázorňují relativní zpomalení vzhledem k testu s nejmenším bufferem (kdy je prakticky zaručeno, že všechna data jsou v cache nalezena).



Obrázek 5.3: Vliv paměti cache na přístup k datům. Na vodorovné ose je velikost použitého bufferu, na svislé je zobrazen čas, který je pro každý počítač relativní vůči době běhu testu s nejmenším bufferem.

Se zvětšováním adresového rozsahu, ze kterého data čteme, se doba běhu nejprve prakticky nemění. To odpovídá situaci, kdy cache pojme veškerá potřebná data. Jakmile překročíme její velikost, roste pravděpodobnost toho, že data v cache nebudou nalezena (cache miss) a je nutné jejich vyzvednutí přímo z operační paměti. Pro velké velikosti vstupního bufferu se pak paměť cache uplatňuje málo a doba běhu testu pak roste již jen mírně. Z bodu zlomu naměřených charakteristik můžeme potvrdit velikost cache L1 u Raspberry Pi 16 KB (vliv L2 není příliš výrazný), u desktopu (Intel Core i7) je zřejmý průdký nárůst křivky v okolí 8 MB odpovídající velikosti L3 cache. K podobnému (byť ne tak prudkému) zpomalení dochází i u serverového procesoru. U něj jsme zaznamenali i částečné odchylky v rozmezí od 128 KB (také rozptyl časů u opakovaných měření byl větší). Možným vysvětlením by mohl být vliv ostatních jader zpracovávajících jiné úlohy (ať již přímý využíváním stejné L3 cache nebo nepřímý díky technologii Turbo Boost, která adaptivně mění taktovací frekvenci). Bez využití výhody cache (tj. pro velké buffery) se doba běhu testu blíží dvojnásobku výchozí hodnoty.

[5]Je třeba poznamenat, že L2 u procesoru BCM2835 není součástí ARM jádra a přístup k ní není přímo řízen zabudovanou MMU. Jedná se o rozšíření firmy Broadcom sloužící v první řadě k podpoře vestavěného GPU. Ve výchozím nastavení současných verzí Raspbianu je podpora L2 již zapnuta, ale obecně platí, že její přínos pro aplikace není tak výrazný jako u jiných procesorů (v některých speciálních případech může být až negativní).

Persistentní úložiště

K uchovávání dlouhodobějších informací potřebuje každý počítač persistentní úložiště. Jeho rychlost významně ovlivňuje dobu potřebnou pro spuštění programů a hlavně pro práci s většími objemy dat. Testovaná zařízení se velmi liší v druhu používaného úložiště, proto je nutné tento stav zohlednit.

	Raspberry Pi B	**Raspberry Pi B+**	**Desktop**	**Server**
rnd_read	0,7	0,7	9,3	5,3
seq_read	17,0	17,3	171,9	404,5
seq_write	2,4	1,7	58,1	86,9

Tabulka 5.2: Persistentní úložiště – přenosové rychlosti v MB/s.

V tabulce 5.2 vidíme dosažené přenosové rychlosti při sekvenčním i náhodném čtení a sekvenčním zápisu. Rozdíly při zápisu mezi oběma modely Raspberry Pi přisuzujeme odlišné použité paměťové kartě. Naměřené rychlosti při sekvenčních testech jsou (v případě zápisu výrazně) nižší než specifikace SD karet, limitujícím faktorem je tedy hardware mikropočítače. U desktopového počítače jsme vzhledem k instalovanému SSD disku naměřili poměrně nízké rychlosti. To může být způsobenu poměrně malým blokem čtených a zapisovaných dat (asi 49 MB), plná rychlost disku by se projevila až při několikanásobně větším objemu. Naopak rychlosti síťového diskového pole u serverového počítače jsou především v sekvenčních testech velmi vysoké a výrazně předčily lokální SSD disk u testovaného desktopu.

Při stavbě počítačového clusteru z Raspberry Pi je vhodné kromě SD (respektive microSD) karet na jednotlivých uzlech použít ještě další perzistentní úložiště připojené přes síťový adaptér. Při dlouhodobém provozu by se mohlo negativně projevit opotřebení buněk karet při zápisu. V těchto případech se doporučuje provoz se systémovým oddílem v režimu pouze pro čtení, přičemž další proměnná systémová (např. logy) i uživatelská data jsou umístěna na síťovém úložišti. Výhodami jsou také odstranění duplicit u sdílených dat a možnost přímého přístupu k datům z vnějšku clusteru. Kromě toho, zápis přes síťové rozhraní je značně rychlejší (viz níže).

Síťový subsystém

Měření latence a propustnosti síťového subsystému probíhalo pouze na Raspberry Pi, protože tyto parametry jsou zajímavé z hlediska propojení těchto počítačů do výpočetního clusteru pomocí sítě. Dosažené hodnoty jsou v tabulce 5.3. Uvedené názvy odpovídají popisu v sekci 4.4.9, tedy v případě testů propustnosti obsahují i velikost pracovního bufferu v kilobytech.

Oba modely Raspberry Pi opět dosáhly díky stejnému hardwaru prakticky totožných výsledků. Hodnoty latence představují horní hranici, protože z podstaty měření jsme dostali součet latence Raspberry Pi a pomocného počítače, který odpovídal na síťové dotazy. Vzhledem k výkonnějšímu hardware druhého počítače předpokládáme jeho vlastní latenci za prakticky zanedbatelnou. Při obdobném testu mezi dvěma uzly clusteru Iridis-Pi (viz 1.5) zjistili autoři latenci uzlu 0,5 ms (jako polovinu celkové latence), což odpovídá našim výsledkům.

Propustnost síťového adaptéru roste s velikostí paketů. Za pozornost stojí vyšší dosažená přenosová rychlost ve spojovaném režimu. Tento jev přikládáme

	model B	model B+
latency	0,561	0,571
tcp_2	34,6	34,1
tcp_8	65,1	64,6
tcp_32	73,0	72,2
udp_2	27,0	26,9
udp_8	52,0	51,7
udp_32	71,4	71,4

Tabulka 5.3: Parametry síťového subsystému – latence v milisekundách a testy propustnosti v Mbit/s.

způsobu implementace síťových protokolů v linuxovém jádru, kde TCP je optimalizován na vysokou propustnost a UDP spíše na nízkou dobu latence. Další rozdíl mezi oběma protokoly jsou poměry využití procesoru – pro 32 KB bloky bylo průměrné vytížení procesoru 99 % při TCP přenosu a pouze 47 % při posílání UDP paketů (obdobně i pro ostatní velikosti bloku). Spojovaný protokol TCP je tedy náročnější na výpočetní výkon, ale v našem testu dosahoval mírně vyšších přenosových rychlostí.

Při testech provádíme střídavě posílání a přijímání dat po blocích dané velikosti (pracujeme vlastně v poloduplexním režimu). Přesto jsou naše výsledky dobře srovnatelné s jinými údaji, např. P. Vouzis naměřil na TCP pomocí nástroje `iperf`[6] rychlost vysílání i příjmu 73,0 Mbit/s [46], maximální propustnost na Iridis-Pi při velkých blocích (od 100 KB) je uváděna 74,1 Mbit/s [24]. Při stahování velkého souboru v lokální síti protokolem HTTP (pomocí programu `wget`) jsme naměřili až 71 Mbit/s. Je nutno poznamenat, že všechny uvedené přenosové rychlosti se týkají skutečně využitelných dat, režie síťových protokolů do nich není započtena. Z našich měření tedy plyne, že v reálných situacích lze využít přibližně tři čtvrtiny teoretické maximální propustnosti síťového adaptéru.

5.3 Pořizovací náklady

Každý z testovaných počítačů míří primárně do jiného segmentu trhu. Je zásadní rozdíl mezi nákupem Raspberry Pi na e-shopu a pořízením serveru u autorizovaného prodejce (navíc často v rámci kompletního řešení informačního systému a individuální cenové nabídky). Další obtíže při porovnání pořizovacích nákladů přináší také změny cen výrobků či kurzů měn. Proto je nutno chápat následující čísla pouze jako orientační.

U Raspberry Pi budeme počítat s cenou 1 400 Kč, která zahrnuje také napájecí zdroj a paměťovou kartu, přičemž cena je shodná pro oba modely. Testovaný desktop je možné pořídit přibližně za 20 000 Kč. U serveru budeme v dalším textu uvažovat pořizovací částku 400 000 Kč. Ta přitom zahrnuje jednak cenu vlastního serveru, dále také část ceny skříně (blade chassis), ve které je server umístěn, a další poprodejní služby.

U zvažovaného clusteru započteme dodatečné náklady na propojení jednotlivých uzlů sítí Ethernet částkou 200 Kč na uzel. Mimo naše úvahy zůstávají další

[6]Iperf – The TCP/UDP Bandwidth Measurent Tool, domovská stránka projektu je na adrese `http://iperf.fr`

položky jako dodatečná síťová infrastruktura, disková úložiště, klimatizace, místo v serverovně a jiné, které jsou společné pro všechna testovaná zařízení.

U Raspberry Pi a desktopového počítače jsme neuvažovali cenu za jejich údržbu a opravy, jelikož neznáme poruchovost těchto zařízení. Havárie serverového počítače pokrývá pětiletá záruka Dell NBD, kdy je zajištěna oprava technikem přímo na místě do druhého pracovního dne.

5.4 Celkové zhodnocení

V předcházejících kapitolách jsme shrnuli výsledky porovnání testovaných zařízení z různých hledisek – spotřeby, výpočetního výkonu a pořizovací ceny. Nyní provedeme srovnání na základě všech tří hlavních kritérií současně.

Z našich měření výpočetního výkonu při aplikačních testech určíme přibližný počet jednotek Raspberry Pi, které by v clusteru dosáhly srovnatelných výsledků jako referenční počítače. Toto číslo získáme jako průměr z reprezentativních hodnot jednotlivých modulů aplikačních testů. Reprezentativní hodnotou modulu rozumíme průměr počtů potřebných Raspberry Pi k zachování stejného výkonu vůči desktopu či serveru dosažených všemi obsaženými testy modulu. Následně pak můžeme porovnávat i pořizovací náklady a spotřebu elektrické energie clusteru a ostatních počítačů.

Pokud přijmeme zjednodušující předpoklad ideální škálovatelnosti výpočetního výkonu mezi jádry procesorů i uzly clusteru, lze výkon referenčního desktopu srovnat se 120 jednotkami Raspberry Pi a výkon serveru pak odpovídá 580 jednotkám. Další údaje lze nalézt v tabulkách 5.4 a 5.5. Zde je nutné zdůraznit, jak problematické je vyjádřit poměr výkonu jedním číslem i jaké nejistoty dosahují naše vstupní údaje pro spotřebu i cenu. Všechny tyto skutečnosti jsou podrobněji diskutovány v předchozím textu.

	Jednotek	**Cena**	**Spotřeba**
Desktop	1	20 000 Kč	250 W
Cluster RPi B	120	192 000 Kč	396 W
Cluster RPi B+	120	192 000 Kč	264 W

Tabulka 5.4: Porovnání parametrů desktopového počítače a clusteru z Raspberry Pi obdobného výpočetního výkonu.

	Jednotek	**Cena**	**Spotřeba**
Server	1	400 000 Kč	500 W
Cluster RPi B	580	928 000 Kč	1 914 W
Cluster RPi B+	580	928 000 Kč	1 276 W

Tabulka 5.5: Porovnání parametrů serverového počítače a clusteru z Raspberry Pi obdobného výpočetního výkonu.

Z uvedených tabulek vyplývá, že postavit výpočetní cluster je obecně dražší než pořízení desktopového nebo serverového počítače odpovídajícího výkonu a také jeho spotřeba energie je vyšší. Nicméně výsledky některých testů vychází částečně pozitivněji z hlediska stavby clusteru. Nejlépe dopadly algoritmy z modulů **sort** a **join**, kde je pro dosažení výkonu desktopového počítače potřeba pouze 35

jednotek Raspberry Pi, respektive 175 jednotek na dosažení srovnatelného výkonu s testovaným serverem. Naopak nejhorší výsledek jsme naměřili u algoritmů z modulu **matrix**, kde by bylo potřeba 410, respektive 2010 počítačů Raspberry Pi.

Naše výsledky pro Raspberry Pi model B jsme porovnali s údaji pro cluster Iridis-Pi, který byl postaven na univerzitě v Southamptonu ve Velké Británii (viz 1.5). Provedená měření se shodují v oblasti pořizovacích nákladů a spotřeby. Nejpodstatnější údaj, totiž výpočetní výkon, přímé srovnání neumožňuje, neboť uváděné hodnoty se týkají pouze benchmarků LINPACK a HPL. Přesto však poměr výkonu Raspberry Pi ku testovanému serveru v této práci řádově odpovídá poměru mezi Raspberry Pi a počítačem s procesorem řady Intel Xeon, na kterém autoři Iridis-Pi provedli měření uvedenými benchmarky a jehož výsledky ve svém článku taktéž zveřejnili.

Ze získaných údajů plyne, že použití staršího modelu B oproti B+ nepřináší žádné výhody. Obecně lze říci, že stavba výpočetního clusteru se zatím nevyplatí. Největšími problémy jsou vyšší pořizovací cena i celková spotřeba proti desktopovým i serverovým počítačům. Nicméně pro vhodné typy úloh lze dosáhnout lepších výsledků ve spotřebě energie i pořizovací ceně než referenční server při zachování výpočetního výkonu. Navíc mezi další výhody clusteru patří také vysoká robustnost či snažší odvod přebytečného tepla.

Myšlence stavby výpočetního clusteru z důvodu úspory nákladů nahrávají dvě události z nedávné doby. První z nich je oznámení o snížení ceny Raspberry Pi modelu B+ o 10 $, druhá (zásadnější) je uvedení nového modelu na trh. Raspberry Pi 2 model B obsahuje čtyřjádrový procesor s 1 GB operační paměti a podle předběžných měření je 3 až 6 krát výkonnější [47] než předchozí model. Cena zůstala zachována na původní hladině a spotřeba by měla odpovídat staršímu modelu B. Podle těchto odhadů se stavba clusteru z nového modelu Raspberry Pi stává výhodnější než testované alternativy i pro obecnější úlohy. Pro přesnější závěry by však bylo nutné provést další měření nového zařízení.

Závěr

V této práci jsme porovnali základní vlastnosti běžně dostupných jednodeskových počítačů. Podrobněji jsme se zaměřili na dvě verze Raspberry Pi a přinesli příklady jejich využití. Dále jsme vysvětlili důležitost distribuovaných výpočtů a zjišťovali, zda je výhodnější postavit výpočetní cluster z těchto malých zařízení nebo koupit běžný serverový či desktopový počítač. V úvahu jsme při tom brali nejen výpočetní výkon, ale také pořizovací a provozní náklady. K porovnání výpočetního výkonu počítačů jsme po analýze dostupných softwarových nástrojů vytvořili metodiku vhodnou především pro podobná jednodesková zařízení, jejíž součástí je také vlastní benchmark. Tím jsme splnili vytyčené cíle a přinesli rozšiřitelný základ pro budoucí testování nejen jednodeskových, ale také desktopových a serverových počítačů.

Po zhodnocení získaných údajů se ukazuje, že uvažovaný cluster je vhodný jen pro některé typy úloh nebo pro specifická místa nasazení vyžadující vysokou robustnost, malé rozměry či snadný odvod přebytečného tepla. Představení nového modelu Raspberry Pi 2 může znamenat zlepšení ekonomické bilance ve prospěch clusteru i u obecnějších úloh, proto plánujeme provést měření i na tomto zařízení.

Použitý benchmark je lehce přenositelný mezi různými počítači podporujícími operační systém GNU/Linux a díky své vhodně navržené architektuře lze snadno přidávat další testovací moduly. Toho mohou využít navazující práce, které pouze rozšíří stávající metodiku a benchmark o potřebné testy a provedou měření s následným vyhodnocením dosažených výsledků na dalších počítačích. Jeden z navrhovaných nových modulů, který nebyl pro jednojádrové procesory testovaných Raspberry Pi zapotřebí, by se mohl zabývat výkonem vícevláknových úloh, protože ty mohou přinést na novějších jednodeskových počítačích vybavených vícejádrovými procesory (včetně Raspberry Pi 2) lepší využití dostupných prostředků. Dalšími kandidáty na rozšíření benchmarku jsou testy grafických čipů nebo dalšího speciálního hardwaru jako je Epiphany koprocesor nebo programovatelná hradlová pole (FPGA) jednodeskového počítače Parallella.

Seznam použité literatury

[1] *Raspberry Pi* [online]. [cit. 2015-05-13]. Dostupné z: http://www.raspberrypi.org/.

[2] *RPi Hardware - eLinux.org* [online]. [cit. 2014-10-24]. Dostupné z: http://elinux.org/RPi_Hardware.

[3] *Představení UniPi Board* [online]. [cit. 2014-10-25]. Dostupné z: http://www.raspi.cz/2014/07/predstaveni-nove-univerzalni-desky-unipi-board/.

[4] *raspberrypi/hats · GitHub* [online]. [cit. 2014-10-25]. Dostupné z: https://github.com/raspberrypi/hats.

[5] *FrontPage - Raspbian* [online]. [cit. 2014-10-25]. Dostupné z: http://www.raspbian.org/.

[6] *Raspberry Pi - Wikipedia, the free encyclopedia* [online]. [cit. 2014-10-25]. Dostupné z: http://en.wikipedia.org/wiki/Raspberry_Pi#Software.

[7] *Wolfram + Raspberry Pi Project: A Wolfram Engine on Every Raspberry Pi* [online]. [cit. 2014-10-25]. Dostupné z: http://www.wolfram.com/raspberry-pi/.

[8] *Minecraft: Pi Edition — Minecraft: Pi Edition updates and downloads* [online]. [cit. 2014-10-25]. Dostupné z: http://pi.minecraft.net/.

[9] *Apollo-NG - PiGI - Raspberry Pi Geiger-Müller Interface* [online]. [cit. 2014-10-26]. Dostupné z: https://apollo.open-resource.org/lab:pigi.

[10] Akerman, Dave. *PIE1 – Raspberry Pi Sends Live Images from Near Space — Dave Akerman* [online]. [cit. 2014-10-26]. Dostupné z: http://www.daveakerman.com/?p=592.

[11] *AirPi Kit v1.4 - Raspberry Pi weather station shield from tmhrtly on Tindie* [online]. [cit. 2014-10-26]. Dostupné z: https://www.tindie.com/products/tmhrtly/airpi-kit/.

[12] *WeeWX: Linux weather software* [online]. [cit. 2014-10-26]. Dostupné z: http://www.weewx.com/.

[13] *PiFace — I/O Interface for Raspberry Pi* [online]. [cit. 2014-10-26]. Dostupné z: http://pifacedigital.wordpress.com/.

[14] *owenquad - Raspberry Pi based quadcopter with full walkthrough coding tutorial - Google Project Hosting* [online]. [cit. 2014-10-26]. Dostupné z: https://code.google.com/p/owenquad/.

[15] *Gizmo 2 — GizmoSphere* [online]. [cit. 2014-11-17]. Dostupné z: http://www.gizmosphere.org/products/gizmo-2/.

[16] *Arduino - Home* [online]. [cit. 2014-10-26]. Dostupné z: http://www.arduino.cc/

[17] *ArndaleBoard.org* [online]. [cit. 2014-10-26]. Dostupné z: http://www.arndaleboard.org/wiki/index.php/Main_Page

[18] *Banana Pi - A Highend Single-Board Computer* [online]. [cit. 2014-10-26]. Dostupné z: http://www.bananapi.org/

[19] *BeagleBoard.org - community supported open hardware computers for making* [online]. [cit. 2014-10-26]. Dostupné z: http://beagleboard.org/

[20] *Intel announces Edison: a 22nm dual-core PC the size of an SD card* [online]. [cit. 2014-10-26]. Dostupné z: http://www.engadget.com/2014/01/06/intel-edison/

[21] *Intel® Galileo Gen 2 Development Board—Empower Your Prototype* [online]. [cit. 2014-10-26]. Dostupné z: http://www.intel.com/content/www/us/en/do-it-yourself/galileo-maker-quark-board.html

[22] *Parallella Computer Specifications — Parallella* [online]. [cit. 2014-10-26]. Dostupné z: http://www.parallella.org/board/

[23] Kiepert, Joshua. *Creating a Raspberry Pi-Based Beowulf Cluster* [online]. [cit. 2014-11-17]. Dostupné z: http://coen.boisestate.edu/ece/files/2013/05/Creating.a.Raspberry.Pi-Based.Beowulf.Cluster_v2.pdf

[24] Cox, Simon J. – Cox, James T. – Boardman, Richard P. – Johnston, Steven J. – Scott, Mark – O'Brien, Neil S. *Iridis-pi: a low-cost, compact demonstration cluster* [online]. Červen 2013 [cit. 2014-11-17]. Dostupné z: http://www.southampton.ac.uk/~sjc/raspberrypi/raspberry_pi_iridis_lego_supercomputer_paper_cox_Jun2013.pdf

[25] Patterson, David A. – Hennessy, John L. *Computer Organization and Design: The Hardware/Software Interface*. 4. vydání. Elsevier, 2012. ISBN 978-0-12-374750-1.

[26] *Intel® Core™ Processors Technical Resources* [online]. [cit. 2014-12-25]. Dostupné z: http://www.intel.com/content/www/us/en/processors/core/CoreTechnicalResources.html#Previous

[27] John, Lizy Kurian – Eeckhout, Lieven. *Performance Evaluation and Benchmarking*. CRC Press, 2005. ISBN 978-1-42-003742-5.

[28] *Phoronix Test Suite - Linux Testing & Benchmarking Platform, Automated Testing, Open-Source Benchmarking* [online]. [cit. 2014-11-12]. Dostupné z: http://www.phoronix-test-suite.com/

[29] *Bonnie++ now at 1.03e (last version before 2.0)!* [online]. [cit. 2014-11-12]. Dostupné z: http://www.coker.com.au/bonnie++/

[30] *SPEC - Standard Performance Evaluation Corporation* [online]. [cit. 2014-11-12]. Dostupné z: http://spec.org/

[31] *LMbench - Tools for Performance Analysis* [online]. [cit. 2014-11-12]. Dostupné z: http://www.bitmover.com/lmbench/

[32] Dongarra, Jack J. – Liszczek, Piotr – Petitet, Antoine. The LINPACK Benchmark: past, present and future. *Concurrency and Computation: Practice and Experience*, 2003, vol. 15, no. 9, s. 803-820. ISSN: 1532-0634.

[33] *Resources / Frequently Asked Questions — TOP500 Supercomputer Sites* [online]. [cit. 2015-04-12]. Dostupné z: http://www.top500.org/resources/frequently-asked-questions/

[34] Daemen, Joan – Rijmen, Vincent. *AES Proposal: Rijndael* [online]. [cit. 2015-03-12]. Dostupné z: http://csrc.nist.gov/archive/aes/rijndael/Rijndael-ammended.pdf

[35] *Descriptions of SHA-256, SHA-384, and SHA-512* [online]. [cit. 2015-03-12]. Dostupné z: http://csrc.nist.gov/groups/STM/cavp/documents/shs/sha256-384-512.pdf

[36] Percival, Colin. *STRONGER KEY DERIVATION VIA SEQUENTIAL MEMORY-HARD FUNCTIONS* [online]. [cit. 2015-03-12]. Dostupné z: http://www.tarsnap.com/scrypt/scrypt.pdf

[37] Dijkstra, E. W. *A note on two problems in connexion with graphs*. 1. vydání. Springer-Verlag, 1959. ISSN: 269-271.

[38] Wagner, Robert A. – Fischer, Michael J. *The String-to-String Correction Problem*. New York, USA: ACM, 1974. ISSN: 0004-5411.

[39] Levenshtein, Vladimir I. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 1966, vol. 10, no. 8, s. 707-710. ISSN: 1028-3358.

[40] *Strassen Formulas – from Wolfram MathWorld* [online]. [cit. 2015-03-10]. Dostupné z: http://mathworld.wolfram.com/StrassenFormulas.html

[41] Deutsch, P. *DEFLATE Compressed Data Format Specification version 1.3* [online]. 1996 [cit. 2015-04-22]. Dostupné z: https://www.ietf.org/rfc/rfc1951.txt

[42] Ziv, Jacob – Lempel, Abraham. A universal algorithm for sequential data compression. *Information Theory, IEEE Transactions on*, 1977, vol. 23, no. 3, s. 337-343. ISSN: 0018-9448.

[43] Huffmann, D. A. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE*, 1952, vol. 40, no. 9, s. 1098-1101. ISSN: 0096-8390.

[44] *zlib Home Site* [online]. [cit. 2015-03-11]. Dostupné z: http://zlib.net/

[45] Bray, Timothy. *The JavaScript Object Notation (JSON) Data Interchange Format* [online]. 2014 [cit. 2015-04-25]. Dostupné z: http://tools.ietf.org/html/rfc7159

[46] VOUZIS, P. *Raspberry Pi and Distributed Network Monitoring: Iperf* [online]. Srpen 2014 [cit. 2015-05-11]. Dostupné z: http://netbeez.net/2014/08/19/raspberry-pi-and-distributed-network-monitoring-iperf/

[47] *Turbocharged quad-core Raspberry Pi 2 unleashed, global geekgasm likely • The Register* [online]. Únor 2015 [cit. 2015-05-13]. Dostupné z: http://www.theregister.co.uk/2015/02/02/raspberry_pi_model_2/?mt=1422862779867.

A. Vzorové výstupy

A.1 Hlavní program

Příklad 1

```
$ ./benchmark -lv -i crypt,mem -c ../config.json
```

spustí program `benchmark` v módu s rozšířeným výpisem, tj. budou vypsány jednotlivé testy modulů a konfigurační parametry. Zahrnuty budou pouze moduly `crypt` a `mem`. Bude použit konfigurační soubor `config.json` v nadřazeném adresáři.

Ukázkový výstup:

```
Benchmark utility, Petr Stefan 2014
Using config file ../config.json
  parsing successful
------------------------------------------------------------------------
Module                                                  File
------------------------------------------------------------------------
  crypt - cryptographic algorithms                      bm_crypt.so
     # aes - AES (Rijndael) block cipher with 256-bit key
     # sha256 - hashing function from SHA-2 family
     # scrypt - key derivation function with high resource usage
     * Params:
         aes buffer size: 8388608B
         aes seed: 1536
         sha256 text: The quick brown fox jumps over the lazy dog.
         sha256 buf size: 16777216B
         sha256 hash size: 32
         scrypt N: 16384
         scrypt r: 8
         scrypt p: 1
         scrypt key size: 64
         scrypt text: pleaseletmein
         scrypt salt: SodiumChloride
  mem - sequential memory access                        bm_mem.so
     # read_1b - sequential memory read in 1B blocks
     # read_2b - sequential memory read in 2B blocks
     # read_4b - sequential memory read in 4B blocks
     # read_8b - sequential memory read in 8B blocks
     # write_1b - sequential memory write in 1B blocks
     # write_2b - sequential memory write in 2B blocks
     # write_4b - sequential memory write in 4B blocks
     # write_8b - sequential memory write in 8B blocks
     * Params:
         memory size: 134217728B
         read loop: 6
         write loop: 6
------------------------------------------------------------------------
```

Příklad 2

```
$ ./benchmark --exclude net,mem,card,cache --times=2
```

spustí testy všech modulů kromě modulu `net`, `mem`, `card` a `cache` s krátkým výpisem. Každý test poběží dvakrát za sebou.

Ukázkový výstup:

```
Benchmark utility, Petr Stefan 2014
Config file not found. Using default values.
aes 0.157309870 0.157200137
aes 0.143516245 0.143447975
sha256 0.088802711 0.088753211
sha256 0.089191121 0.089143166
scrypt 0.096577911 0.096504137
scrypt 0.096115013 0.096047694
multiply_4 0.022833923 0.022822420
multiply_4 0.022617914 0.022606185
multiply_8 0.027534580 0.027519842
multiply_8 0.027608007 0.027591479
strassen_4 0.028177454 0.028134703
strassen_4 0.028517904 0.028473959
strassen_8 0.030291520 0.030252664
strassen_8 0.031564896 0.031522611
quicksort 0.082895293 0.082807705
quicksort 0.083616394 0.083547283
merge_1 0.034152339 0.034124470
merge_1 0.033661527 0.033633045
merge_2 0.042641756 0.042610674
merge_2 0.042377866 0.042346831
hash_1 0.034419833 0.034332984
hash_1 0.034307898 0.034230757
hash_2 0.036093929 0.036002923
hash_2 0.035845823 0.035762003
levenshtein 0.028084336 0.028025202
levenshtein 0.024360646 0.024292635
zlib 0.162685772 0.162545226
zlib 0.162716924 0.162561484
dijk_fast 0.002152368 0.002149613
dijk_fast 0.002195238 0.002192476
```

A.2 Net-echo

```
$ ./net-echo
```

spustí pomocný síťový odpovídající program pro testy z modulu `net` s výchozími parametry.

Ukázkový výstup:

```
Network echo for benchmark (TCP/UDP, IPv4/IPv6)
  listening on TCP port 10000
  listening on UDP port 10000
Waiting for connection ... (Ctrl-C to quit)
Connection detected ...
  on UDP port from node [::ffff:127.0.0.1]:40643
  262144 bytes echoed
Waiting for connection ... (Ctrl-C to quit)
Connection detected ...
  on TCP port from node [::ffff:127.0.0.1]:49779
  8388608 bytes echoed
Waiting for connection ... (Ctrl-C to quit)
```

B. Výsledky testů

V následujících tabulkách jsou uvedeny zpracované výsledky pro všechny testované počítače. Ke všem testům je uveden aritmetický průměr a směrodatná odchylka pro reálný i procesorový čas, vždy uváděny v sekundách.

Raspberry Pi

	model B		**model B+**	
Test	**Reálný čas**	**Procesorový čas**	**Reálný čas**	**Procesorový čas**
aes	2,35740 ± 0,00147	2,35135 ± 0,00067	2,35825 ± 0,00185	2,35159 ± 0,00065
cache	11,66320 ± 0,00330	11,63283 ± 0,00254	11,73703 ± 0,00524	11,70150 ± 0,00344
dijk_fast	5,51562 ± 0,27235	5,50098 ± 0,27148	5,53307 ± 0,14554	5,50507 ± 0,12373
hash_1	3,20201 ± 0,00332	3,19387 ± 0,00323	3,20837 ± 0,00038	3,20033 ± 0,00033
hash_2	3,37652 ± 0,00426	3,36866 ± 0,00461	3,38392 ± 0,00099	3,37505 ± 0,00063
latency	2,29985 ± 0,02439	1,00970 ± 0,00207	2,33827 ± 0,04094	1,02557 ± 0,00204
levenshtein	2,10447 ± 0,00109	2,09890 ± 0,00076	2,10321 ± 0,00123	2,09720 ± 0,00048
merge_1	1,36397 ± 0,00111	1,36032 ± 0,00058	1,36310 ± 0,00098	1,35959 ± 0,00054
merge_2	2,47218 ± 0,00131	2,46582 ± 0,00070	2,47109 ± 0,00098	2,46423 ± 0,00060
multiply_4	33,20860 ± 0,00334	33,14209 ± 0,00243	33,11557 ± 0,00336	33,04310 ± 0,00203
multiply_8	44,77443 ± 0,00781	44,68629 ± 0,00797	44,69801 ± 0,01765	44,60366 ± 0,01696
quicksort	1,90792 ± 0,00073	1,90273 ± 0,00041	1,90758 ± 0,00091	1,90211 ± 0,00091
read_1b	11,84649 ± 1,05834	11,80043 ± 1,04946	13,23395 ± 0,07647	13,18800 ± 0,06496
read_2b	8,38560 ± 0,83570	8,35282 ± 0,82036	9,13356 ± 0,04620	9,09767 ± 0,04422
read_4b	6,17924 ± 0,57069	6,16373 ± 0,56929	6,54221 ± 0,46934	6,52460 ± 0,46780
read_8b	5,97728 ± 0,00075	5,96234 ± 0,00058	5,97661 ± 0,00134	5,96079 ± 0,00072
rnd_read	66,03961 ± 0,43187	56,54234 ± 0,31614	66,06826 ± 1,56638	56,52178 ± 1,55113
scrypt	1,24372 ± 0,00101	1,24084 ± 0,00056	1,24210 ± 0,00111	1,23900 ± 0,00050
seq_read	2,86854 ± 0,00072	0,32435 ± 0,00120	2,82350 ± 0,00589	0,32782 ± 0,00090
seq_write	20,28976 ± 0,92072	1,60116 ± 0,00716	28,10000 ± 0,77357	1,75443 ± 0,00255
sha256	1,07410 ± 0,00020	1,07184 ± 0,00012	1,07394 ± 0,00035	1,07145 ± 0,00017
strassen_4	6,21863 ± 0,00093	6,20303 ± 0,00093	6,31280 ± 0,00104	6,29604 ± 0,00102
strassen_8	8,50531 ± 0,00213	8,48406 ± 0,00196	8,71318 ± 0,00214	8,68993 ± 0,00138
tcp_2	3,88236 ± 0,85227	2,60334 ± 0,08292	3,94085 ± 0,82916	2,63226 ± 0,09493
tcp_8	2,06309 ± 0,01187	2,02224 ± 0,00868	2,07777 ± 0,02166	2,03978 ± 0,01443
tcp_32	1,83972 ± 0,00535	1,81455 ± 0,00549	1,85862 ± 0,01226	1,83333 ± 0,01107
udp_2	4,96347 ± 0,03723	1,83400 ± 0,00184	4,99154 ± 0,04592	1,81405 ± 0,00305
udp_8	2,58215 ± 0,00508	1,04860 ± 0,00336	2,59358 ± 0,00451	1,05186 ± 0,00187
udp_32	1,87852 ± 0,00232	0,88125 ± 0,00079	1,87945 ± 0,00231	0,88799 ± 0,00207
write_1b	8,64874 ± 0,00146	8,63179 ± 0,00069	8,65309 ± 0,00120	8,63493 ± 0,00044
write_2b	2,33565 ± 0,00109	2,33072 ± 0,00042	2,33614 ± 0,00112	2,33095 ± 0,00049
write_4b	2,33593 ± 0,00125	2,33102 ± 0,00053	2,33610 ± 0,00110	2,33075 ± 0,00047
write_8b	1,28805 ± 0,00093	1,28513 ± 0,00028	1,28852 ± 0,00104	1,28556 ± 0,00049
zlib	3,99022 ± 0,00202	3,98019 ± 0,00172	3,91583 ± 0,00187	3,90528 ± 0,00169

Referenční desktopový a serverový počítač

	Desktop		Server	
Test	**Reálný čas**	**Procesorový čas**	**Reálný čas**	**Procesorový čas**
aes	0,12988 ± 0,00239	0,12986 ± 0,00239	0,20443 ± 0,00024	0,20431 ± 0,00024
cache	0,27418 ± 0,00008	0,27415 ± 0,00008	0,29764 ± 0,00058	0,29734 ± 0,00058
dijk_fast	0,25642 ± 0,00142	0,25640 ± 0,00144	0,36260 ± 0,00370	0,36238 ± 0,00371
hash_1	0,20122 ± 0,00172	0,20117 ± 0,00167	0,32989 ± 0,00171	0,32991 ± 0,00191
hash_2	0,19978 ± 0,00200	0,19976 ± 0,00201	0,33245 ± 0,00179	0,33217 ± 0,00161
levenshtein	0,13343 ± 0,00181	0,13341 ± 0,00183	0,23697 ± 0,00033	0,23682 ± 0,00032
merge_1	0,16146 ± 0,00031	0,16143 ± 0,00032	0,25636 ± 0,00136	0,25619 ± 0,00137
merge_2	0,19738 ± 0,00348	0,19735 ± 0,00348	0,31149 ± 0,00108	0,31130 ± 0,00108
multiply_4	0,22817 ± 0,00009	0,22815 ± 0,00009	0,37533 ± 0,00019	0,37511 ± 0,00019
multiply_8	0,22688 ± 0,00012	0,22683 ± 0,00007	0,37702 ± 0,00024	0,37679 ± 0,00023
quicksort	0,22140 ± 0,00087	0,22137 ± 0,00088	0,35200 ± 0,00171	0,35180 ± 0,00170
read_1b	0,46719 ± 0,00821	0,46714 ± 0,00822	0,78225 ± 0,00128	0,78176 ± 0,00127
read_2b	0,23919 ± 0,00410	0,23916 ± 0,00410	0,41108 ± 0,00170	0,41081 ± 0,00170
read_4b	0,12881 ± 0,00009	0,12880 ± 0,00009	0,30058 ± 0,00120	0,30039 ± 0,00120
read_8b	0,07271 ± 0,00074	0,07271 ± 0,00073	0,35003 ± 0,00168	0,34981 ± 0,00168
rnd_read	5,25461 ± 0,04982	4,86420 ± 0,03509	9,19843 ± 0,37197	8,37942 ± 0,37704
scrypt	0,07897 ± 0,00007	0,07896 ± 0,00006	0,12717 ± 0,00029	0,12709 ± 0,00029
seq_read	0,28406 ± 0,06891	0,02088 ± 0,00018	0,12073 ± 0,00607	0,04145 ± 0,00120
seq_write	0,84021 ± 0,04418	0,20739 ± 0,02315	0,56210 ± 0,01458	0,10934 ± 0,00096
sha256	0,09689 ± 0,00002	0,09688 ± 0,00002	0,14868 ± 0,00021	0,14858 ± 0,00021
strassen_4	0,20529 ± 0,00378	0,20527 ± 0,00379	0,31988 ± 0,00036	0,31969 ± 0,00037
strassen_8	0,22532 ± 0,00386	0,22531 ± 0,00385	0,35058 ± 0,00019	0,35036 ± 0,00018
write_1b	0,47121 ± 0,00820	0,47115 ± 0,00822	0,76005 ± 0,00029	0,75958 ± 0,00030
write_2b	0,12873 ± 0,00089	0,12871 ± 0,00089	0,29292 ± 0,00027	0,29273 ± 0,00027
write_4b	0,12865 ± 0,00093	0,12863 ± 0,00092	0,29289 ± 0,00025	0,29271 ± 0,00025
write_8b	0,13172 ± 0,00100	0,13170 ± 0,00099	0,29963 ± 0,00017	0,29944 ± 0,00016
zlib	0,17377 ± 0,00016	0,17371 ± 0,00009	0,24713 ± 0,00027	0,24699 ± 0,00026