Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék

Bálint Áron

JELFELDOLGOZÓ ALGORITMUSOK HATÉKONYSÁGÁNAK VIZSGÁLATA

Jelfeldolgozás különböző SBC platformon

KONZULENS

Dr. Orosz György

BUDAPEST, 2019

Tartalomjegyzék

1 Bevezetés.....7
1.1 Egykártyás számítógépek általánosságban.....7
1.2 Lehetőségek feltérképezése.....7
1.3 Miért pont a Raspberry Pi.....9
1.4 Konfiguráció.....10
2 A mérési és feldolgozási környezet előkészítése.....12
2.1 USB hangkártya.....12
2.2 Real-time működés.....13
2.3 Kernel és real-time ütemező.....14
2.4 Saját 64 bites kernel fordítása.....15
2.5 Operációs rendszer konfigurálása.....18
2.6 Tesztjel.....19
3 Teljesítménymérés.....20
3.1 PMU számlálók.....20
3.2 Perf.....20
3.3 Mérés folyamata.....22
4 Optimalizálási eszközök.....29
4.1 SIMD.....29
4.2 Advanced SIMD technology (Neon technology).....29
4.3 Rust.....30
5 FIR szűrők.....32
5.1 Általános ismertető.....32
5.2 Implementációk C nyelven.....34
5.3 Implementációk Rust nyelven.....39
6 IIR szűrők.....41
6.1 Általános ismertető.....41
6.2 Implementációk C nyelven.....43
6.3 Implementációk Rust nyelven.....45
7 Mérési eredmények.....46
7.1 Kernel és izolált magok vizsgálata.....46

7.2 Izolált mag és nem izolált mag stressz hatására....48
7.3 Implementációk összehasonlítása....49
7.4 Iterációk és ismétlődések vizsgálata....55
7.5 Mérések különböző fokszámokkal....57
7.6 Mérések különböző számábrázolásokkal....60
7.7 Gyorsítótár hatásának vizsgálata....64
8 Valósidejű használat....65
8.1 Adatok gyűjtése....66
8.2 Adatok feldolgozása....68
8.3 Adatok szinkronizálása....71
8.4 Konklúzió....73
9 Értékelés és kitekintés....79

HALLGATÓI NYILATKOZAT

Alulírott **Bálint Áron**, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2019. 12. 13.

...…….
Bálint Áron

Összefoglaló

A szakdolgozat során az Olvasó megismerkedhet az egykártyás számítógépekkel, azok csoportosításával és felhasználási lehetőségükkel. Kiemelt szerepet kapott ebben a Raspberry Pi 3B+ modell, amin méréseim jelentős részét és a jelfeldolgozást végzem el.

A felhasznált eszközök megismerése után röviden betekintést nyerhet az Olvasó a real-time ütemezővel ellátott kernel fordításának lépéseibe és az operációs rendszer felkonfigurálásba a mérésekhez.

Bemutatok egy lehetséges módot Linux disztribúciókon futó programok teljesítményének mérésére hardveres számlálók és a perf eszköz segítségével. Az Olvasó betekintést nyerhet egy méréseket automatizáló Bash Shell szkriptbe, amivel lehetséges különböző mérési környezetet és munkaterületet létrehozni az éppen mérendő programnak.

A szakdolgozat során különböző szűrőalgoritmusokat valósítok meg C és Rust nyelven az elmélet rövid ismertetése után. A definíció alapú megvalósítás mellett olyan technológiákkal implementálom az algoritmusokat, amik lehetővé teszik az adatok vektoros kezelését, így akár az optimalizálást is.

A mérések során elsősorban az egyes implementációk közötti különbségeket, a szűrő fokszámfüggőségét mérem le, de kitérek a gyorsítótár hatására is. Ezek mellett foglalkozok számábrázolási kísérletekkel is.

A jelfeldolgozás során egy egyszerű programkeretet mutatok be lejátszó és felvevő eszközök használatával, ahol a feldolgozó algoritmus és a felvevő eszköz cserélhető, akár szimulálható is. A feldolgozás során a bejövő adatokat alakítom át és külső hangkártya segítségével a külvilág felé továbbítom a szűrt jelet.

Abstract

This thesis describes briefly the types and use cases of single board computers, especially Raspberry Pi 3B+, which was used at most of the performance measuring and signal processing in my thesis.

To measure the possible impact of real-time kernels, the flow of kernel building is described along with the configuration of the operation system setting.

I've used the "perf" linux tool for performance monitoring which is capable of using the built-in Performance Monitor Units. Perf creates a fairly readable input for each iteration, but it's important to compare the whole measurement too. Each measurement was done in its own environment which was managed and automated by a Bash Shell script. It handles the measurement specific parameters, such as CPU frequency, cache and temperature.

The other half of the paper describes how FIR and IIR filters can be implemented and measured. For implementation I used C and Rust, but the signal processing is only implemented in C. Besides the basic implementation of filters I've used technologies which allow to use less instructions for more data and to exploit vectorization and optimization.

I was interested at most in the differences of implementations and the tap-dependency of filters, but some other measurements were done too. For example, the impact of cache and temperature changing was measured too.

The signal processing of float numbers is implemented in C. The recorder collects data for the processing thread, which sends the processed data to the DAC module. The record device can be changed at compile time to a built-in square wave, random numbers or a simple file too in case we do not have an ADC.

1 Bevezetés

1.1 Egykártyás számítógépek általánosságban

Egykártyás számítógépek, szakirodalomban Single Board Computers, olyan kis méretű számítógépek, amelyeknek működésükhöz szükséges összes elemük egy nyomtatott áramköri lemezre van integrálva. Olcsó áruk is ennek köszönhető. Bővítésük gyártóspecifikus, gyakran úgynevezett „shieldeket” adnak ki, amelyek további modulokat tartalmaznak. Általánosságban beágyazott rendszerként és oktatási eszközként szokták használni. Kettő típusa terjedt el [1]:

- **Traditional SBC:** Hagyományos egykártyás számítógépek, amelyek valamilyen ipari szabvány alapján épültek meg. Gyakran magas számú általános célú be- és kimenettel rendelkeznek (GPIO) és gyártótól függően alkalmazásspecifikus kiegészítő modulokat támogatnak.
- **COM-Based SBC:** Computer-on-module esetén a CPU egy külön kártyán van, így cserélhető komponens a számítógépben. Sokkal nagyobb rugalmasságot nyújt a célközönségnek egy kettő kártyából álló SCB, mint egy tradicionális. Előnye a skálázható teljesítmény és a hosszabb élettartam a cserélhetőség miatt, mivel az újabb CPU csere nem okoz gondot. Hátránya, hogy kompatibilisnek kell lennie a kettő kártyának.

1.2 Lehetőségek feltérképezése

Olyan egykártyás számítógépeket kerestem, amik képesek valamilyen Linux disztribúciót futtatni, sok változatuk van, amelyek egymással összehasonlíthatóak és Magyarországon is könnyen beszerezhetőek. Előny, ha a számítógéphez kapható valamilyen DAC modul vagy kiegészítő, illetve rendelkezik rendes dokumentációval vagy felhasználói tábor által működtetett fórummal.

1.2.1 Raspberry Pi

A Raspberry Pi (gyakran említve: Pi) egy hagyományos egykártyás számítógép, ami olcsó árának és támogatottságának köszönhetően széles körben elterjedt a világon.

Az oktatástól kezdve a mesterséges intelligenciáig mindenhol használják diákok és mérnökök is. Ennek köszönhetően rengeteg alkalmazásspecifikus kiegészítővel rendelkezik, hatalmas felhasználótáborral rendelkező fóruma van, továbbá temérdek blog található hobbiprojektekről, amikből kiválóan lehet betekintést nyerni új területekre és más szemszögéből megismerni a Pi alapjait. [2]

1.2.2 Banana Pi

A Banana Pi is egy Arm alapú nyílt-forráskódú hardver projekt, rengetegféle kártyával. A kártyák teljesen kompatibilisek a Pi kiegészítőivel, még a Raspberry-re készült Linux képfájlok is futtathatóak rajtuk. Magyarországon is könnyen beszerezhetőek. [3]

1.2.3 BeagleBoard

A BeagleBoard egy alacsony energiaigényű nyílt-forráskódú egykártyás számítógép a Texas Instruments cégtől. A kártya eredetileg oktatókártyának készült egyetemek számára. A BeagleBoard-ok az Arm architektúrát követik, így számos Linux telepíthető a kártyára, még Android is. A kártyák rendelkeznek külön DSP chipekkel, amik a hang és videó feldolgozást segítik. Jelenleg több alkalmazásfüggő változata is elérhető kiegészítőkkel, viszont hivatalos DAC modul jelenleg nincsen. [4]

1.2.4 Onion Omega2

Az Onion Omega2 család kis méretű, olcsó IoT-ra specializált számítógépekből áll, 580 MHz-es MIPS CPU-val vezérelve. Képesek Linux disztribúciókat futtatni, számos kiegészítő modullal rendelkezik és hatalmas dokumentáció áll a kezdő felhasználó részére. Az eszközöket IoT használatra tervezték, ebből adódik kis méretük és energiatakarékosságuk, valamint Ethernet és WiFi csatlakozási lehetőségük is. A hozzátartozó parancssoros operációs rendszer „soft real-time” és több programozási nyelvet is támogat, mint például a C, Python és a Node.js. Legnagyobb különbség közte és a Rasberry család között az energiafelhasználása, valamint alkalmatlansága a teljesítményigényesebb alkalmazásoknak, mint például a HD videó lejátszása. Hátránya még, hogy Magyarországon nehezen elérhetőek a kártyák, valamint a hivatalos kiegészítők között nincsen DAC. Az Arduino Dock kiegészítő segítségével az Arduino „shieldeket” lehet hozzá illeszteni, amik ezt a hiány pótolhatják. [5]

1.2.5 Intel Galileo

Az Intel Galileo a 32 bites Intel x86 architektúrára épít. Ez a számítógép kifejezetten a hobbi projekteknek és oktatásra készült. Jelenleg kettő változata van: Gen1 és Gen2. Szoftveresen és hardveresen is teljesen kompatibilis az egyik legnépszerűbb Arduino mikrokontrollerrel, az Arduino Uno-val, viszont már nem forgalmazzák. Közvetlenül programozható az Arduino IDE-ből és kompatibilis a kiegészítőivel is. [6]

1.3 Miért pont a Raspberry Pi

A választásom azért esett a Pi családra, mert könnyedén lehet hozzájutni Magyarországon is mind a számítógépeikhez, mind a kiegészítőikhez. A feladatom során főleg Pi 3B+ számítógépet használok egy USB-s hangkártyával. A 16 bites lebegőpontos számok vizsgálatánál a 3B+ modell mellett Pi 1 és Zero modelleket is alkalmazok.

Elterjedtségéből adódóan a felhasználók több operációs rendszer közül választhatnak szándékuk és ismereteik szerint. A hivatalos operációs rendszer a Raspbian, ami egy Debian alapú Linux disztribúció. Feladatom során én viszont nem csak ezt használom, mivel a jelenlegi Raspbian 32 bites és csak ArmV7 architektúrát támogat. ArmV8-tól kezdődően viszont újabb lehetőségek nyíltak a jelfeldolgozás számára. Lehetséges Raspbian esetében is kihasználni a 64 bites ARM architektúrát 64 bites kernellel, de egyszerűbbnek láttam Ubuntu Server használatát, ami 64 bites operációs rendszer 64 bites kernellel. [7] Célnak megfelelően - például IoT, médiaközpont - különböző, erre specializálódott operációs rendszereket is telepíthetünk rá. Az Arm processzoroknak köszönhetően akár még Android operációs rendszert is telepíthetnek a felhasználók.

1.3.1 Főbb különbségek

A Raspberry Pi modellek jelentősen eltérnek egymástól, mégis hasonlóak, így több forrásból is lehet a teljesítménybeli különbségeket elemezni. Az általam lényegesnek talált és elemzett szempontok: utasítások száma, ciklusok és futási idő.

A Pi 3B+ modellben négy darab Arm Cortex-A53 található, Armv8-A architektúrával. Mivel több maggal rendelkezik, elkülöníthetünk egy magot csak a

feldolgozó algoritmusnak, hogy más „user-space” folyamat ne zavarhassa meg. Az utasításkészlet kaput nyit Neon utasítások használatára: 128 bites vektorregiszterek lehetőséget adnak néhány jelfeldolgozó algoritmus optimalizálására vektorizálással, vagyis vektorként kezeljük azokat az adatokat, amiket egyébként skalárként használnánk. Megjelentek a VFPv4 FPU-k, amik Neon támogatottságúak: vektorműveleteket lehet elvégezni akár lebegőpontos számokon is. ArmV8 architektúrának köszönhetően megjelentek a 64 bites alkalmazások, így az operációs rendszer megválasztására is figyelni kell a Pi 3B+ esetén. Másik oldalról a Pi Zero processzora egy régebbi és egyszerűbb, ARM1176JZF-S, ami 32 bites Arm11 RISC processzor lebegőpontos társprocesszorral.

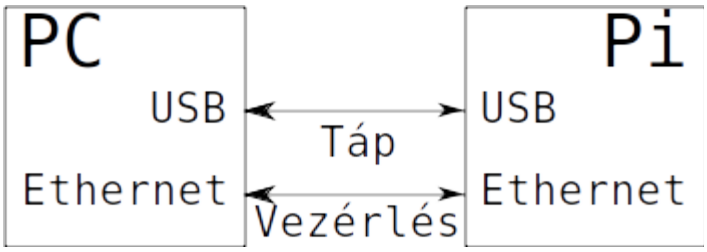
A lebegőpontos társprocesszor segítségével az eszköz képes gyorsan és hatékonyan lebegőpontos számokkal műveletet elvégezni. Jelfeldolgozás során ez kifejezetten előnyös lehet, sok jelfeldolgozó algoritmus dolgozik lebegőpontos számokkal. Amennyiben nem rendelkezünk ilyen alegységgel, használhatunk fixpontos törtszámábrázolást a műveleteinkhez. Ekkor az egész-rész és a törtrész pontosságát a bitek aránya határozza meg, melyik részhez mennyi bitet rendelünk hozzá. A számot egész számként tároljuk, így nincsen szükség lebegőpontos műveletekre. Előnye a lebegőpontos számábrázolással szemben, hogy van bitszintű reprezentációja C nyelvben. Lebegőpontos számoknak alapvetően nincsen, de például egy azonos méretű egésszel és egy *Union* típusú változóval lehet kezelni hasonlóan. A szakdolgozatom során kihasználom a FPU jelenlétét és lebegőpontos számokkal végzem el a szükséges műveleteket.

1.4 Konfiguráció

1.4.1 Raspberry Pi 3B+

Ubuntu Server 18.04. LTS Linux disztribúciót futtat az eszköz, minimális alkalmazásokkal és csomagokkal. A Pi és a hoszt közötti kapcsolat Ethernet segítségével történik SSH-n keresztül, tápot közvetlenül a gépemtől USB-microUSB kábelen keresztül kap. Grafikus felületet nem tartalmaz, a szakdolgozat során nincsen rá szükség, feleslegesen venné el mástól az erőforrást.

A 64 bites processzorának köszönhetően lehetőség van 64 bites operációs rendszert és kernelt futtatni rajta. Szakdolgozatom során az eszközön megvizsgálom, hogy milyen hatással lehet a mérésekre egy real-time kernel.



1. Ábra: A Raspberry Pi és a PC kapcsolata

1.4.2 Raspberry Pi 2, 1B és Zero

A Pi 1 és Zero modellekhez Raspbian Buster Lite-ot használok 4.9-es kernellel, Pi 2 modellhez az előzőekben ismertetett Ubuntu Linuxot. A Pi 1 kártya rendelkezik Ethernet porttal, ezért a hoszttal való kommunikációra Ethernet kapcsolatot használok. Zero esetén Ethernet port hiányában microUSB-Ethernet átalakítóval csatlakoztatom a hoszthoz. Első indításnál az SSH kapcsolat biztonsági okokból le van tiltva, ezért engedélyezni kell úgy, hogy egy üres „ssh” nevű fájlt hozunk létre a kernel fájlok mellé. Az első bejelentkezés után az SSH-t engedélyezni kell a későbbi használatra.

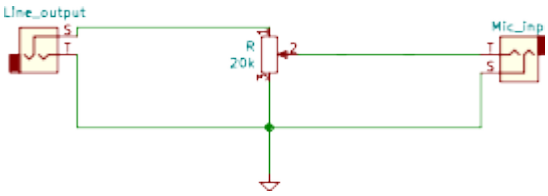
WiFi kapcsolat hiányában a Raspberry Pi 1-en telepítem fel a rendszer csomagjait és konfigurálom be. Ezek után csinálok egy új képfájlt, amit bármikor vissza tudok állítani a későbbiekben kísérletezések során azért, hogy ne kelljen mindig elvégezni ezeket a lépéseket.

2 A mérési és feldolgozási környezet előkészítése

2.1 USB hangkártya

A jelfeldolgozás során szükségem lesz egy bemeneti forrásra, amiből az érkező adatokat feldolgozom és kiadom a hangkártyán. Erre a célra egy egyszerű 16 bites USB hangkártyát használok, ami rendelkezik 3,5 mm-es Jack mikrofon bemenettel és fejhallgató kimenettel.

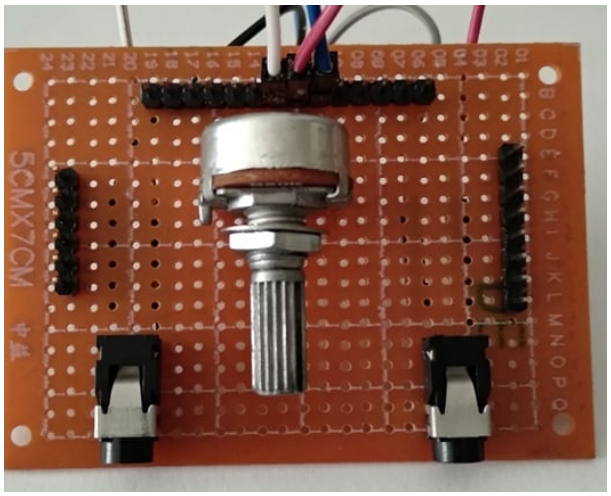
A jel forrása egy másik eszköz lesz, ahonnan a fejhallgató kimeneten keresztül küldöm a jelet a mikrofon bemenetnek. Mivel a kimenet („line level”) nagyjából 1 V és a bemenet („mic level”) nagyjából 1 mV, ezért a kimeneti jelszintet a bemenetihez kell igazítani. [8] A jelszintet egy potenciométer segítségével igazítom a bemeneti jelszinthez.



2. Ábra: Jelforrás illesztése a bemenethez

Jelenlétét a Linux hangkártyakezelőjének (ALSA) egyik parancsával tudjuk kijelezni:

```
Input: arecord -l
card 1: Device [USB Audio Device], device 0: USB Audio [USB Audio]
  Subdevices: 1/1
  Subdevice #0: subdevice #0
```



3. Ábra: Az áramkör prototípusa

2.2 Real-time működés

Real-time működés alatt azt értjük, hogy az eszköz egy adott kérésre adott határidőn belül valamilyen egyhez közeli valószínűséggel válaszolni fog. Eme valószínűség alapján kettő kategóriát szoktunk megkülönböztetni: „soft” és „hard real-time”. „Soft real-time” esetén a kérésre adott válasz mulasztása nem jár kritikus következményekkel, valamint a valószínűség arra, hogy a válasz időben érkezik, közel egy. Ezzel szemben, biztonságkritikus alkalmazásoknál megkövetelt a „hard real-time”, ahol a valószínűség annyira megközelíti az egyet, amennyire csak lehetséges. Az általam kiválasztott Linux disztribúció nem képes „hard real-time” működésre, csak kifejezetten az erre készített operációs rendszerek. Érdemes megemlíteni, hogy létezik közösség által „átportolt” FreeRTOS, viszont a „hard real-time” működéshez megfelelő szoftver is kell, önmagában nem tudja biztosítani. Rendszeres frissítés hiányában a Ubuntu Server Linux disztribúciónál maradtam.

Két lehetséges módja van annak, hogy egy ilyen rendszert készítsünk: törekszünk arra, hogy a modellünk a lehető legpontosabb legyen hardveresen és szoftveresen, vagy megfelelően ütemezzük a feladatainkat. Kész hardver lévén én a modell pontosságát a kernel lecserélésével próbálom javítani, valamint az ütemezőt is a feladatnak megfelelően választom ki. [9]

2.3 Kernel és real-time ütemező

A kernel az operációs rendszer magja, amit a bootloader tölt be a memóriába minden indításkor. A hardver erőforrásait kezeli és nyilvántartja, az összes I/O kérés rendszerhívásként rajta megy keresztül, minden támogatott driver nála található meg a felhasználótól elrejtve. Interfészt biztosít a felhasználói alkalmazások és a hardver között. A kernel privilégium szintet is jelöl, megkülönböztetjük a kernel folyamatokat a felhasználói folyamatoktól. Többféle kialakítása létezik, hármat különböztetünk meg [10]:

- **monolitikus**: minden szükséges drivert az éppen aktuálisan használt kernel tartalmaz modulként, ha éppen hiányzik valami belőle, akkor bele kell fordítani. Hibás modul esetén teljes összeomlást is okozhat. A driverek növekvő számával a kódméret is fokozatosan nő. Ilyenek például a Linux kernelek.
- **mikrokernel**: a kernel csak a kritikus dolgokkal foglalkozik: a memóriával és a processzorral. A driverek nem részei a kernelnek, így hibás driver esetén nem omlik össze a kernel. Könnyebb kezelhetőséget ad, valamint a kódméret is sokkal kisebb a monolitikushoz képest.
- **hibrid:** a monolitikus és a mikrokernel előnyeit kombináló kernel. Az operációs rendszer összes szolgáltatása a kernelben üzemel. Ezzel megszűnik a felhasználói tér megbízhatósági előnye, valamint a felhasználói tér, kernel tér közötti kontextusváltás költsége is csökken.

Ütemező az operációs rendszer folyamatainak (process) szálait (thread) ütemezi és szabályozza, hogy éppen mikor melyik kaphat futáshoz jogot vagy processzort, mint erőforrást. Egy ütemező preemptív, ha az operációs rendszer képes elvenni a processzort egy száltól, azt váró állapotba helyezni és a futáshoz való jogot egy másik szálnak átadni. Ekkor kontextusváltás történik (szakirodalomban context switch, process switch vagy task switch), miközben a folyamathoz tartozó környezetet az operációs rendszer kimenti és betölti a másik folyamathoz tartozó környezetet kernel módban. Lényegében a kernel ilyenkor felfüggeszti egy feladatnak a futását és átadja az erőforrást egy másiknak. Egy folyamat vagy szál környezetén az ahhoz tartozó regisztereket, mint például az utasításszámlálót, stack pointert, általános regisztereket értjük és emiatt költséges műveletnek is tekinthető. Általában a preemptív ütemezők prioritásos alapon dolgoznak. [11]

Érdemes különbséget tenni a preemptív ütemező és a real-time működés között. A preemptív ütemező nem jelenti azt, hogy a működés real-time, ha a kernel nem real-time.

Sem az Ubuntu Server, sem a Raspbian alapesetben nem egy valósidejű operációs rendszer (real-time operating system, RTOS), mert a kernel nem real-time, annak ellenére, hogy vannak „patchek”, amik hozzáadnak olyan képességet, hogy néhány feladat real-time ütemezhető legyen, de a Linux fővonalába ezek nem tartoznak bele. A szakdolgozatban megvizsgálom, mekkora jelentősége van a méréseket elvégezni valósidejű és nem valósidejű operációs rendszer alatt Raspberry Pi 3B+ eszközön.

2.4 Saját 64 bites kernel fordítása

Kernel készítésére található leírás a Raspberry hivatalos oldalán [12], bárki használhat egyedi beállításokkal saját kernelt. A kernelt Ubuntu 19.04 hoszton fordítottam Bash Shell használatával (cross compiling), nem a Pi-n önmagán. Utóbbi is lehetséges lenne, viszont egy nagyságrenddel lassabb lenne a folyamat.

A kernelt Raspberry Pi 3B+ számítógépre fordítom, mivel csak ennek a processzora támogatja a 64 bites architektúrát. Raspberry Pi 4-es modellek is támogatják, viszont nem áll rendelkezésemre ilyen eszköz.

Nulladik lépésként kell egy tetszőleges Linux kernel, amihez van „real time patch” vagy választhatunk olyat is amiben már benne van. [13]

Első lépésként megnéztem az eredeti kernelt, mert számít a fájl neve a későbbiekben. A fájlnév alapján beállítottam egy környezeti változót, ami megegyezik az eredetinek a nevével, illetve egy kimeneti mappát ami megkülönbözteti a fordított kerneljeimet.

```
export KERNEL=kernel8
export OUTDIR=out-$(date '+%H%M')
```

Következő lépésként meg kell mondani a fordítónak, hogy milyen alapkonfigurációt használjunk. A Raspbian kerneléből kikerestem a Pi-hez tartozó alapbeállításokat (bcmrpi3_defconfig) és a továbbiakban ezt használtam kisebb módosításokkal. Ezután a fordítás a következőképpen zajlik: minden fordítási lépésben megadjuk, hogy melyik mappába fordítunk, milyen architektúrára (arm64) és azt, hogy

mivel (aarch64-linux-gnu-gcc-8). Ezzel létrehozunk az eggyel feljebbi mappában egy kimeneti mappát, ahol az összes kernel fájl lesz.

```
make O=../${OUTDIR}/ ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu-
bcmrpi3_defconfig
```

Következő lépésben beállítjuk a saját beállításainkat. Törekedtem arra, hogy egy minimális kernelt kapjak, amiben ne legyen olyan funkció amit nem fogok használni. Ehhez remek segítségét ad a beépített „menuconfig”, ami parancssorból futtatható és a beállítások elvégzését segíti: hierarchikusan mutatja a beállítandó pontokat és kérésre bővebb leírást ad az éppen kijelölt pontról. Utóbbi kifejezetten hasznos volt, de nem mindig nyújtott elegendő információt ahhoz, hogy milyen hatással lenne a kernel teljes egészére és el tudjam dönteni, hogy biztonságosan ki- vagy bekapcsolhatóak-e a szolgáltatások és funkciók.

```
make O=../${OUTDIR}/ ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu-
menuconfig
```

Első dolgom az ütemező beállítása volt, amiért elsősorban új kernelt akartam fordítani. Az általános beállítások alatt lehet megtalálni a menüpontot és három lehetőség közül lehet választani. Én a legszigorúbbat, az erősen preemptív változatot jelöltem be.

Ügyeltem arra, hogy engedélyezve legyenek a PMU számlálók (Performance Monitor Unit, hardveres teljesítmény vizsgáló egység), mivel a későbbiekben ezek használatára építek. A beállítások elvégzése után kaptam egy olyan kernelt, ami képes több magot ütemezni akár preemptíven. A virtualizálást kikapcsoltam, hiszen nem tervezek a Pi-n virtuális gépet futtatni, valamint a CAN busz, NFC, TV és „Amatőr Rádió” szolgáltatásokat (AM/FM) is kivettem a kernelből az FPGA driverekkel együtt. Érdekességként, a beállítások között meg lehet adni, hogy a fordító teljesítményre vagy méretre optimalizáljon. Itt az alap beállítást hagytam meg, miszerint minél jobb teljesítményre törekszik a compiler. Ahhoz, hogy később el tudjam különíteni a magokat, az SMT ütemezés támogatását engedélyezni kell.

Konfiguráció mentése után megkezdődhet a fordítás. Szemléletesebb parancsában megadjuk azt a három dolgot ami szükséges a kernelnek:

1. **Image**: (32 bites Pi esetében zImage) a kernel képfájl, ami be lesz töltve a memóriába indításkor.

2. **Modules**: Ezeket a modulokat a kernel fordítása után telepíteni kell külön az SD kártyára. Ügyelni kell itt, hogy megadjuk az SD kártya helyét, különben a saját hosztunkra telepítené alapesetben. Ilyen modul például a hangkártyákhoz tartozó driverek, amik nélkül a rendszer nem tudná használni az illesztett eszközöket.
3. **DTB**: Device Tree Blob, egy adatbázis amiben a hardver komponensek szerepelnek. Ezzel ad át a bootloader alacsony szintű hardver információkat a kernelnek. [14]

```
make -j4 O=../${OUTDIR}/ ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu-
Image modules dtbs
```

A parancs hatására elkezdődik a fordítási folyamat. Amennyiben minden jól ment, nem fogad minket hibaüzenet a legvégén és feltelepíthetjük a microSD kártyára a modulokat és DTB-ket. Itt nem szabad elfelejteni megadni azt, hogy hova telepítsük, különben a hosztra próbálja meg telepíteni.

```
sudo env PATH=$PATH make O=../$OUTDIR/ ARCH=arm64
CROSS_COMPILE=aarch64-linux-gnu-
INSTALL_MOD_PATH=/media/aron/writable modules_install dtbs_install
```

Végül fel kell másolni a most készített képfájlt és „dtb” fájlokat az eszközre. Érdemes a régit külön elmenteni, hogy meglegyen biztonsági mentésként hibás kernel esetén.

```
sudo cp /media/aron/system-boot/$KERNEL.img /media/aron/system-boot/
$KERNEL-${OUTDIR}.img

sudo cp ../$OUTDIR/arch/arm64/boot/Image
/media/aron/system-boot/$KERNEL.img

sudo cp ../$OUTDIR/arch/arm64/boot/dts/broadcom/*.dtb
/media/aron/system-boot/

sudo cp ../$OUTDIR/arch/arm64/boot/dts/broadcom/*.dtb
/media/aron/system-boot/broadcom

sudo cp ../$OUTDIR/arch/arm64/boot/dts/overlays/*.dtb*
/media/aron/system-boot/overlays/
```

Ha minden jól megy, akkor visszahelyezve a microSD kártyát a Pi-be, a következő indításnál már az új kernel fogad minket. Új kernel verzió ellenőrzése:

```
Input: $ uname -a
Output: Linux ubuntu 4.19.71-rt24-v8 #1 SMP PREEMPT RT Tue Sep 24
08:53:07 CEST 2019 aarch64 aarch64 aarch64 GNU/Linux
```

Itt láthatjuk, hogy tényleg real-time preemptív (PREEMPT RT) 64 bites kernelt kaptunk (aarch64).

Tapasztalatok:

Vannak kernelspecifikus csomagok, ilyen a *linux-tools* csomagban a *perf*, amit én az algoritmusok teljesítményének vizsgálatára használok. Ezt a kernel eszközt is manuálisan kellett fordítani, mivel kernelspecifikus eszköz. Kényelmi okokból ezt már közvetlenül a Pi-n fordítottam a függőségek telepítése után. Még egy változtatást végeztem a fordítás előtt: mivel „csv” (coma separated values) kimenetet dolgozok fel saját Python szkripttel és alapértelmezetten a „csv” kimenetre nem íródik ki a folyamat által használt idő, ezért a forrásban engedélyeztem a kiírást. A patch a függelékben megtalálható ehhez.

A fordítás folyamán szempont volt, hogy a magokat el lehessen különíteni egymástól, egy felhasználói folyamatot hozzá lehessen rögzíteni. Ehhez az SMP támogatást engedélyezni kellett. Későbbiekben a real-time patchet tartalmazó kernellel nem sikerült elérnem a magok izolálását ezzel sem, viszont a beállítás szükséges volt a patchet nem tartalmazó kernel esetén.

Az általam használt egyik kernel verzió végül 4.19.71-rt24-v8 lett Ubuntu Server disztribúcióra. A kernel frissítése és a csomagok telepítése után készítettem képfájlokat a microSD kártyán található partíciókról, hogy a későbbiekben kísérletezés után könnyedén vissza tudjak állni egy már jól működő változatra.

2.5 Operációs rendszer konfigurálása

A rendszer tesztelése során ügyelni kell arra, hogy ne rakjunk fel olyan csomagokat, amik függőségként egy újabb kernelt is telepítene mellé. Előfordulhat ilyenkor olyan eset, hogy újraindítás után az operációs rendszer nem indul el és hibaüzenetet sem ad. Tekintettel arra, hogy a kernel sem töltődik be ilyenkor, valószínűleg a saját és a csomagok által frissen telepített új kernel nem kompatibilis valamilyen módon.

A mérések során szükségem volt különböző fordítókra és perf eszközre, ezért ezeket telepítettem a dependenciáikkal. A feldolgozáshoz és a feldolgozás nyomkövetéséhez az ALSA csomagokra és könyvtárra és a WiringPi könyvtárra van

szükségem, így azokat kell telepíteni. Az egyik mérés során megvizsgáltam, hogy stressz hatására a mérések hogyan torzulnak, ehhez a „stress” csomagot használtam.

Első futtatásnál az SSH-t engedélyezni kell és a szolgáltatást külön el kell indítani, hogy minden újraindításnál elinduljon magától. A későbbiekben így akár hosztról közvetlenül lehet programozni az eszközön.

2.6 Tesztjel

Az algoritmusok működésének ellenőrzésére és teljesítményének mérésére négyszögjelet használok, amivel a szűrő ugrásválaszát meg tudom vizsgálni. A jel és a szűrő koefficiensek időtartománybeli tagjait Octave programmal állítom elő a *signal* csomag segítségével. Periodikus jel lévén elég egy periódust előállítani belőle egységnyi értékekkel és eltárolni a programban.

3 Teljesítménymérés

3.1 PMU számlálók

A Performance Monitoring Unit számlálók speciális programozható számlálóregiszterek, amik teljesítmény monitorázását teszik könnyedén lehetővé. Ezek működése és száma architektúránként eltérőek lehetnek, például az Arm11 és Arm Cortex-A5-ös architektúra esetében kettő PMU számláló van definiálva. [15]

Működésük az egyszerű számlálókon és ezen funkciót támogató assembly utasításokon alapul, amivel kiolvassák a számlálóregiszterből a kívánt adatot és átírják egy másik speciális konfigurációs regiszterbe. Még egy követelményük van, hogy fizikailag össze legyenek kötve a mérendő hardverelemmel. Ezek a számlálók vezérelhetőek, tehát ki- és bekapcsolhatóak. Minden egyes eseményre, amire be vannak programozva a hozzájuk tartozó számláló értéke nő. Érdemes megemlíteni a Performance Monitoring Counter regisztereket (PMC), ami a PMU-hoz hasonlóan működik, viszont vele ellentétben nem programozható, fixen ugyanazokat az adatokat olvassák. [16]

3.2 Perf

Az ismertetőt Brendan D. Gregg munkája [17] és a perf wiki [18] alapján készítettem el. és A perf egy Linux profiler eszköz, ami a linux-tools csomag részeként található meg. A profiler képes elkapni hardver eseményeket, úgynevezett „perf_event” és statisztikát készíteni belőlük. Mindezt úgy képes megvalósítani, hogy a futtatandó bináris állomány módosítatlan marad. Képes a teljes rendszerre is méréseket elvégezni, nem kell feltétlenül állományhoz kötni.

Megkülönböztetjük a szoftveres és hardveres eseményeket. Szoftveres események közé tartozik minden olyan esemény, amelyekhez nem tartozik hardveres számláló. Az ilyen események közvetlenül a kernelből jönnek, például a kontextus váltás. Ezzel szemben a hardveres események hardverrel mérhetőek, például PMU számlálókkal, tipikusan ilyen a ciklusszám és az utasításszám. A kettő kategória között vannak a cache események, amelyeknek csak akkor van értelme, ha az adott

processzoron támogatva van az esemény. Az elérhető és mérhető kategóriákat típustól függetlenül a *perf list* paranccsal hívhatjuk elő.

Az eszköz esetén megkülönböztetünk több módot. A kettő főbb mód ezek közül a számolás (counting) és mintavételezés (sampling). Számlálás során a hardveresemények folyamatosan összegződnek és a program futása után a sztenderd kimeneten megjelenik, amennyiben nem lett átirányítva. Mintavételezés során először a kernel bufferbe gyűjti, majd egy fájlba írja az adatokat szál, folyamat vagy akár CPU alapján. Én a méréseim során a számolás módot használom.

Két lehetséges mód van futtatható állomány vizsgálata: vagy perf segítségével indítunk el egy folyamatot vagy akár hozzá is csatolhatjuk egy már meglévő folyamathoz is.

Példa statisztika: a „Hello Raspberry” kiíratása egy iterációval és iteráción belül tíz ismétléssel. Egy-egy statisztika kulcs-érték párokból és a hozzájuk tartozó kommentekből áll.

```
Performance counter stats for 'echo Hello Raspberry' (10 runs):

900278      cycles:u                                    ( +-  2.62% )
7663293     cycles:k                                    ( +-  0.98% )
352704      instructions:u  #    0.39  insn per cycle   ( +-  0.01% )
5822468     instructions:k #    0.76  insn per cycle    ( +-  0.63% )
8           context-switches                            ( +- 11.37% )
0           cpu-migrations
63          page-faults                                 ( +-  0.32% )
912767      branches                                    ( +-  0.59% )
39550       branch-misses   #    4.33% of all branches  ( +-  1.81% )

0.009708 +- 0.000654 seconds time elapsed  ( +-  6.74% )
```

Első oszlopban a kulcsok értékeit látjuk, második oszlopban a hozzájuk tartozó kulcsot. Ezt opcionálisan egy komment követi. Több mérés esetén zárójelek között egy értékhez viszonyított intervallum található, amiben az érték megtalálható.

Érdekességként megemlíthető, hogy egy egyszerű kiírás is rengeteg ciklusból és utasításból áll. Magyarázat erre az, hogy a kiírás során a sztenderd kimenetre írunk, ami szintén egy fájl. A fájlírások input-output műveletek, amik a kernelen mennek keresztül, ezért lassúak és költségesek. Érdemes még *ltrace* eszközzel is megnézni és láthatjuk, hogy olyan függvényeket is meghívunk, amiket nem is sejtenénk. A „Hello Raspberry” sztringünket összehasonlítjuk az *echo* utasítás lehetséges paramétereivel (--help és –version), amik további utasításokat és ciklusokat adnak bele a statisztikába.

Mezők értelmezése: ami „:u”-ra végződik az „user-space”, ami „:k”-ra végződik az „kernel space” adat. Ahhoz, hogy „kernel space” eredményeket lássunk rendszergazdaként kell elindítanunk a mérést, különben a felhasználói jogokkal rendelkező perf nem tudja kiolvasni a számlálók értékét.

Létezik egy könnyebben feldolgozható változata is, ahol a kulcs és az érték vesszővel, a kulcs-érték párok új sorral vannak elválasztva. Könnyebb feldolgozni, viszont hátránya, hogy a felhasználónak ránézésre kevesebbet árul el.

Egy felvetődő probléma a PMU számlálókkal az eszközfüggőség. Mint ahogy a Pi eszközök is különbözőek, ezért a PMU támogatottságuk is más. Például sem a Raspberry Pi 1-es, sem a Zero v1.3 modell nem támogatja sem az utasításszám, sem a ciklusok mérését, ami megnehezíti az algoritmusok hatékonyságának vizsgálatát.

3.3 Mérés folyamata

A méréseket egy Bash szkript segítségével automatizáltam. A szkript bemenetként kaphat egy iterációszámot, ahányszor kell futtatni a programot és magát futtatandó programot, amit a szkript becsomagol egy környezetbe és átadja perfnek a megfelelő változókkal. Minden mérésnek készítünk egy saját mappát az éppen aktuális dátummal, amibe a mérési eredmények szöveges fájlja bekerül. A méréshez szükséges feltételeket is a szkript valósítja meg: ellenőrzi a Pi CPU hőmérsékletet és csak bizonyos érték alatt kezd meg egy új mérést, amennyiben ezt kérjük a szkripttől, elkülöníti a magokat, hogy legyen egy dedikált mag a mérendő feladatnak, stabilan tartja a frekvenciát és kérésre üríti a gyorsítótárat is.

3.3.1 Munkatér beállítása

A szkript létrehoz egy új mappát és fájlt, amibe a hőmérsékletet írja, mint saját kulcs-érték pár, valamint a perf kimenete is ez a fájl lesz. Minden paraméterbeállítás itt zajlik le: Például a paraméterként átadott iterációszámot és futtatható állományt egy külön változóba menti le, amit a későbbi ciklus során tud használni, valamint a maximális hőmérséklethatár is itt állítható be, ami fölött már nem kezd meg új mérést. A perfnek átadott sztring kategóriákat is ennél a résznél állítom be. Ezek a kulcsszavak alapján dolgozik a perf a későbbiekben.

Kettő mérési egységről lehet beszélni: iterációk és ismétlődések. Iterációk körülölelik az ismétlődéseket és egy-egy iteráció között történhet lehűlés és gyorsítótár törlés. Egy iteráció a futtatható állomány egyszeri futtatását jelöli. Ezzel ellentétben az ismétlődések folyamatosan zajlanak le és két ismétlődés között nem lehet semmi más utasítás. Az iterációk és ismétlődések száma is paraméterként beállítható.

4. Ábra: Mérési egységek. Példaként egy három iterációból álló mérés kettő ismétlődéssel.

3.3.2 Hőmérséklet ellenőrzése

A hőmérséklet ellenőrzésének létjogosultsága kérdéses. Amennyiben van az SBC-ben olyan teljesítményre hatással való elem, ami érzékeny a hőmérséklet-változásra, akkor érdemes vizsgálni és megfelelő határok között tartani a mérési hőmérsékletet. Amennyiben nem rendelkezik ilyen összetevővel, akkor sem árt az eszközt „egészséges”, nem túl forró hőmérsékleten tartani. Magas hőmérséklet esetén a processzor kénytelen visszavenni a frekvenciát alacsonyabb értékre. Az ellenőrzést a ciklus elején teszem meg mindig. Amennyiben nagyobb, mint a megadott érték, nem kezd új mérést a szkript, hanem aktív várakozásba kezd. Másodpercenként lekérdezi az aktuális hőmérsékletet és amennyiben kisebb az aktuális érték, mint a határ, akkor megkezdheti a perf a munkáját.

Lehetőség van a méréseket hiszterézises hőmérséklet-szabályozással megvalósítani. Az elágazások elkerüléséért és az egyszerűség kedvéért az aktuális hőmérséklethatárt a program futása során váltogatom a határok között. A funkció kikapcsolható egy változóval, amennyiben nem akarjuk minden mérésnél korlátozni a

futási hőmérsékletet. Itt feltételezem, hogy nem fogok 0 °C-on hőmérsékletet mérni, ezért ahhoz a hőmérséklethez kötöm a mérés engedélyezését. Pszeudokóddal leírva egyetlen iteráció:

```
Hőmérséklet = Kiolvasás_a_szenzorból()

Ha Maximum nem 0
     Limit = Maximum

     Ciklus amíg a Hőmérséklet nagyobb, mint a Limit:
          Alvás()
          Limit = Minimum
          Hőmérséklet = Kiolvasás_a_szenzorból()
          Írás_naplófájlba(Hőmérséklet)

Mérés()
```

3.3.3 Magok elkülönítése

A leírást a Red Hat cikke alapján készítettem el. [19] Szeretnék elkülöníteni egy teljes magot a perfnek a teljesítménymérésre azért, hogy semmilyen más folyamat ne tudja megzavarni a mérési eredményeket és a lehető legtöbb processzor időt kapja a saját feladatom. Linux ilyenkor eltávolítja az összes „user-space” szálat a magról és azokat a kernel szálakat amiket tudja, áthelyezi egy másikra. A megszakításokat is levehetjük arról a magról, így külső esemény hatására sem zavarhatjuk meg a feladatunk működését.

A magok nem teljesen egyenértékűek, ezért nem mindegy, hogy melyiket választjuk ki a méréshez. Tipikusan a nulladik mag felel a megszakításokért, ezért azt nem célszerű elkülöníteni magunknak. A méréshez én a negyedik magot használtam és különítettem el a többitől a következő módon: az „isolcpus” boot paramétert be kell állítani a kijelölt magokra, ezzel megakadályozzuk, hogy még „boot időben” más szálak használják azokat. Későbbi jelfeldolgozáshoz szükségem lesz még egy izolált magra, ezért a harmadik magot is elkülönítettem. Itt eltértem a leírástól, mert az én esetemben nem változtatta meg közvetlenül a kernel paramétereket, ezért nekem kellett manuálisan. Ezeket a paramétereket kell hozzáadni a *cmdline* fájlhoz a boot partíción.

```
isolated_cores=3,4
nohz=on
nohz_full=3,4
```

A *nohz_full* lehetővé teszi, hogy az egyetlen egy magon futó feladatnak a kernel ne küldjön „CPU tick-et”, kevesebb kontextusváltás legyen, így több ideje maradjon a feladatnak saját futására. Ezzel szemben a *no_hz* beállítása csökkenti a számláló megszakításokat az idle processzoron. Idle állapotban éppen nem végez hasznos feladatot a processzor, csak az idle feladatot.

A Bash szkriptemet csak annyiban kell ezek után módosítani, hogy amikor a perf-et meghívom, akkor a „user-space” folyamatoktól izolált magon fusson. Erre lehetőséget nyújt a *schedtool*: megadjuk neki a futtatandó állományt és a CPU maszkot, ahol minden bit egy-egy CPU-t reprezentál.

A megoldás helyettesíthető, illetve kiegészíthető úgynevezett „cpu shieldekkel” vagy külön a folyamatoknak létrehozott „cpusetekkel”. Utóbbiak strukturáltabb és átláthatóbb szerkezetet adnak, valamint az elkülönített magokhoz a folyamatok dinamikusabban is hozzárendelhetőek, memóriakezelő egységük is kiválasztható, utóbbiból Pi esetén csak egy van.

3.3.4 Frekvencia stabilan tartása

A méréseket érdemes ugyanazon a frekvencián elvégezni, ami természetesen eltérhet eszközönként. A Linux egyik nagy előnye, hogy majdnem minden fájlként is értelmezhető, így az aktuális CPU frekvenciát és módot könnyedén kiolvashatjuk. Azonban a Pi 3B+ csak kétféle frekvenciát képes felvenni: 600 MHz és 1.4 GHz, az alacsonyabb frekvencia lesz a mérési frekvencia. Raspberry Pi Zero esetén ez az érték 700 MHz-re módosul, Pi 2 esetén 600 MHz és 900 MHz értékek érvényesek.

Első lépésként „performance” módba kell állítani az eszközt, ami stabil frekvencián fogja tartani az eszközt. Második lépésként a kívánt frekvenciát kell beállítani, amit a legközelebbi frekvencialépcsőhöz igazítja az eszköz. Ezt tehetjük különböző csomagok segítségével, vagy akár az *echo* paranccsal a megfelelő pszeudo fájlba írunk.

3.3.5 Gyorsítótár kiürítése

3.3.5.1 Warmup iterations

Erre azért lehet szükség, mert gyakran az első pár százaléka a teljes mérésnek felfogható egy tranziens folyamatnak, amikor a hardver még nem szokott hozzá a

feladathoz. A hardvernek általában kell egy kis idő, ameddig tud alkalmazkodni a kapott feladathoz: a gyorsítótárnak idő kell, hogy minden szükséges vonalat beolvasson, illetve azt is ki kell találnia, hogy egyáltalán milyen adatokat és honnan érdemes beolvasni, ez a „cache policy” kérdése. A „warmup iterations” jelenség után a programnak már minden erőforrás teljesen rendelkezésére áll. Ennek párja a „timing iterations”, amikor már a „tranziens” a hardverben végbement, „állandósult állapotba” kerül a hardver. Érdemes megvizsgálni, hogy az én méréseimnél megkülönböztethetők-e ezek a kétféle iterációk és amennyiben igen, mennyire. Még egy érdekes kérdést felvet ennek az iterációtípusnak a létezése, mégpedig, hogy hogyan érdemes mérni? Kell-e minden teszt előtt gyorsítótárat kiüríteni, vagy sem. [21]

3.3.5.2 Gyorsítótár

Mivel a Linux rendszereknél majdnem minden elérhető fájlként, ezért a gyorsítótárat is könnyedén elérhetjük. Biztonsági szempontokból viszont csak rendszergazda módosíthatja a fájl tartalmát, vagyis ürítheti a különböző gyorsítótárakat. Amennyiben engedélyezve van a gyorsítótár kiürítése, bevárunk minden gyorsítótárazott írást, majd kiürítjük a lapgyorsítótárat, amire az egyes érték utal a kiírásban. Funkciót kikapcsolhatóvá tehető egy változóval, hogy lehessen ténylegesen vizsgálni a „warmup iterations” jelenlétét. [22]

3.3.6 Finomhangolás

Miután készítettem egy eszközt arra, hogy a méréseket automatizáljam, érdemesnek találtam a szkriptnek a paraméterezését megcsinálni úgy, hogy a mérések automatizálását is lehessen automatizálni: jelenleg a szkript fix paramétereket fogad *getopt* segítségével. [20] A célom az, hogy ezeket a paramétereket valamennyire dinamikussá tegyem. A feladat egyszerű, mert csak saját paraméterhalmazokkal kell meghívnom többször az automatizáló szkriptet, a szkript megoldja a többit most már.

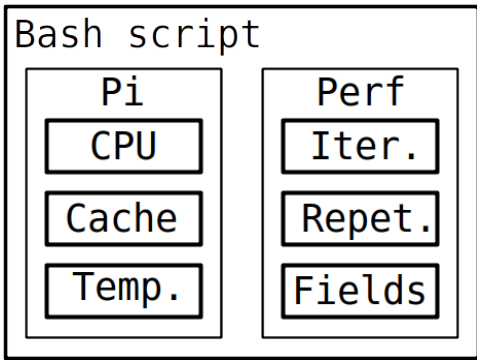
A paramétereket *-h* csatoló futtatásával lehet látni az alapértelmezett mezők értékeivel.

```
-i <number_of_iterations: default ${iterations}>
-r <repetition in the same environment: default ${repetition}
-e <executable: default ${executable}>
-m <minimum_temperature: default ${templimitMIN}>
-M <maximum_temperature: default ${templimitMAX}>
  #Set this to 0 if you want to disable temperature controlling.
```

```
-s <sleep_time_during_cooldown: default: ${sleeptime}>
-c <1:clean cache, 0:use cache; default: ${clean_cache}>
-n <name_of_the_workspace: deafult: the actual date>
-f <fields_for_perf_as_a_string>
-h help
```

3.3.7 Összefoglalás

A Bash szkript begyűjti a paramétereket, majd létrehozza a munkaterületet. A méréseket külön, a mérés dátumával elnevezett mappába menti egyetlen egy fájlba, ez a saját munkaterülete. A perf kategóriáit sztringként kell megadni még a perf futása előtt, különben az alapértelmezett kategóriákat használja. Az eszköz túlmelegedése során a naplófájlba írja a hőmérsékletét egészen addig ameddig folytatni nem tudja a mérést, amennyiben engedélyezzük a hőmérséklet korlátozását. Ez hiszterézisen van megoldva, tehát alacsonyabbra kell lehűlni, mint amennyi a határ a túlmelegedésre. A lapok gyorsítótárának törlése változóval engedélyezhető és tiltható, hogy a „warm up iterations” jelenség vizsgálható legyen. A program iterációnként bővíti a kimeneti fájlt, futás közben is olvashatóak a részeredmények. Az iterációkon belül lehet engedélyezni ismétléseket. Az egyes ismétlődések között nem kérdeződik le a hőmérséklet, a gyorsítótárat sem lehet törölni köztük. Amennyiben értékük egy, nem ismétlődnek és a legnagyobb mérési egység az iteráció lesz.



5. Ábra: A Bash szkript szerkezete

4 Optimalizálási eszközök

A SIMD és a Neon leírását az Arm Neon bevezetője [23] és az Arm Neon Programmer’s Guide alapján [24] készítettem el.

4.1 SIMD

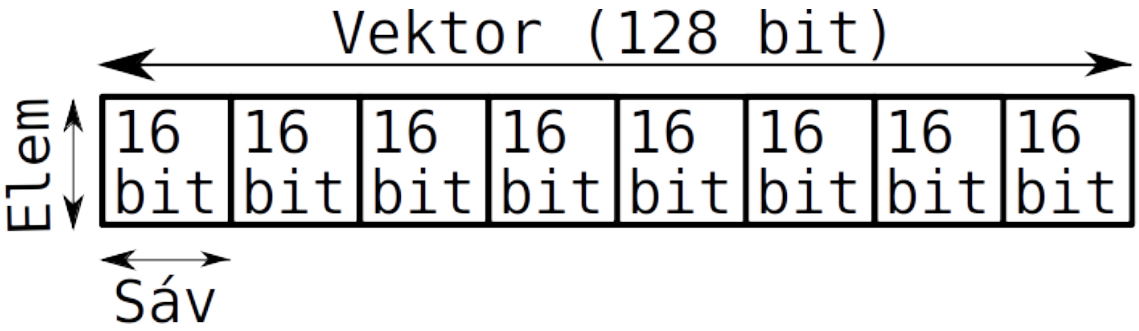
A Single Instruction Multiple Data az adatokon végzett műveletek módját jelöli. Több adaton, vektorokon végzünk el műveletet egyetlen hardveresen támogatott utasítás segítségével, viszont az egyetlen utasítás annyi lépésből áll, ahány elemű a vektorunk. SIMD több architektúrára ki lett fejlesztve és gyakran magasabb szintű nyelvi elemekkel is lehet alkalmazni.

A Raspberry Pi modellek esetében mindegyik modell tartalmaz lehetőséget SIMD utasítások használatára, mert Arm architektúra esetén ArmV6-tól már támogatott 32 bites vektorműveletekre.

4.2 Advanced SIMD technology (Neon technology)

A Neon egy SIMD architekturális kiegészítő Arm Cortex-A processzorokhoz ArmV7-től, amivel jelentősen lehet gyorsítani videó- és hangfeldolgozás hatékonyságát a párhuzamosításnak köszönhetően, amit a vektorműveletekkel megenged. Ezt a technológiát használják arcfelismerésnél és mély tanulásnál is Arm architektúrákon valamint fizikai szimulációkhoz. Hétköznapibb használatuk a 2D és 3D-s alkalmazásokban rejlik.

Működése 32 darab 128 bites regiszteren alapul, amelyek támogatják a sávos adatelérést, valamint a fordítón, ami képes autovektorizálni és optimalizálni a kódunkat. A vektorok sávokra vannak felosztva, amikben elemek találhatóak.



6. Ábra: Egy vektor szerkezete

A „compiler intrinsics” egy olyan függvényhívás, amit a compiler lecserél meghatározott, jelen esetben Neon utasításokra. Így magas szintű nyelvi elemekkel tudunk alacsony szinten programozni. Az assembly nyelven való programozás kiváltható ezek használatával és nagyobb átláthatóságú kódot is kapunk ezáltal. Ezen kívül a fordító gondoskodik a regiszterallokációról és az optimalizálásról is. Használatukhoz a Neonhoz tartozó fejlécet kell beillesztenünk, illetve a 16 bites lebegőpontos számok használatához a saját fejlécüket.

Neon kód fordításához néhány extra paraméter szükséges, amikkel a kívánt funkciókat alkalmazni tudjuk. Példa egy olyan optimalizált fordításra, ahol megengedünk olyan optimalizálásokat amelyek alapból a sztenderdektől eltérhetnek. Az „ffast-math” paraméter a vektorizációt segíti elő. A példában kékkel van jelölve a 16 bites lebegőpontos számok opcionális használatához szükséges rész, zölddel az architektúra specifikus részek:

```
# Raspberry Pi 3B+ modellre
aarch64-linux-gnu-gcc -O2 -ftree-vectorize filter.c -ffast-math -
march=armv8.2-a+fp16 -o filter

# Raspberry Pi 1, 2 és Zero modellre
arm-linux-gnueabihf-gcc -O2 -ftree-vectorize -mfpu=neon filter.c -
ffast-math -mfp16-format=ieee -march=armv6+fp
```

4.3 Rust

A Rust egy C/C++ programozási nyelvhez hasonló nyelv, amivel magas szintű és megbízható szoftvereket lehet készíteni. Lehet közvetlenül teszteket írni a megírt modulokhoz, ezért a teszt-vezérelt fejlesztési modell kiválóan alkalmazható Rust nyelven való programozás során. Rengeteg modullal rendelkezik („crate”), amikkel

újabb funkcionalitásokat adhatunk hozzá a programunkhoz. Egy ilyen „crate” segítségével használhatunk SIMD utasításokat (*packed_simd*), vagy akár Neon utasításokat (*core_arch*), így ki lehet használni a hardver adta lehetőségeket is. Utóbbit módosítani kell az esetünkben, hogy tudjuk használni, ehhez patch a függelékben található. Itt közvetlenül összehasonlítható a Neon és a SIMD implementációk közötti teljesítménykülönbségek. Fordításnál figyelni kell arra, hogy sebesség szerinti optimalizálást válasszuk. Fontos megemlíteni, hogy a SIMD és Neon használata a *nightly* Rusthoz van kötve[1], mert ezek a modulok kísérleti állapotban vannak.

[1]Az általam használt verzió a rustc 1.40.0-nightly (2019-10-04).

5 FIR szűrők

5.1 Általános ismertető

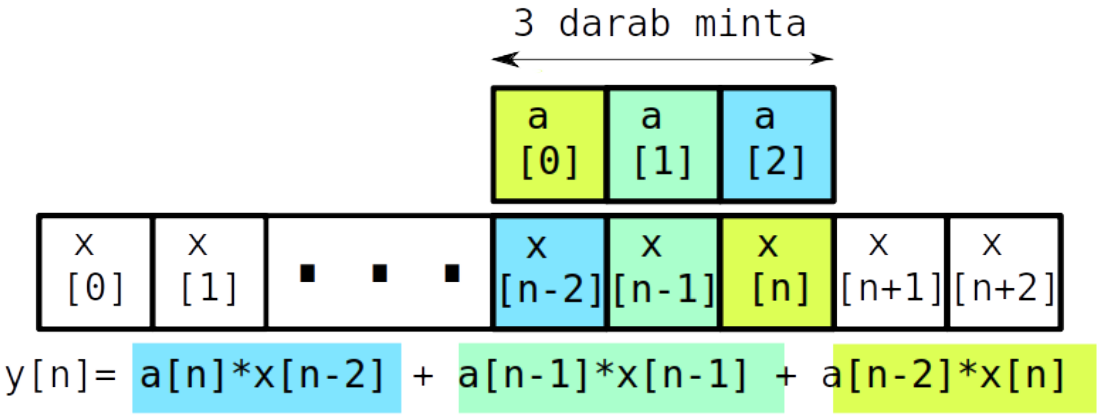
Az ismertetőt a MIT OpenCourseWare [25] leírása és Méréselmélet tankönyv [26] alapján készítettem el. A FIR (finite impulse response, magyarul véges impulzus válasz) szűrő jellegzetessége és egyik nagy előnye, hogy időben véges hosszúságú az impulzusválaszuk, egy idő után teljesen lecseng, zérus értéket vesznek fel a tagok.

$$y(n)=\sum_{k=0}^{N} a(k)\,x(n-k) \tag{1}$$

A fenti képlettel adott mozgó átlag (MA) formulával lehet leírni a FIR szűrők viselkedését, ahol

- y(n) a kimeneti diszkrét értelmezési tartományú szűrt jel
- x(n-k) a bemeneti diszkrét értelmezési tartományú jel az „n-k” időpillanatban
- a(k) bemeneti jel együtthatói (koefficiensek)
- N a szűrő fokszáma (együtthatók száma).

Ez nem azt jelenti, hogy a valóságban nem lehet olyan FIR szűrő, amiben van visszacsatolás, csak nem tipikus. A lecsengő tulajdonság a szűrő struktúrájára utal. Egy másodfokú szűrő esetén a következőt kapjuk a konvolúció kibontása során:

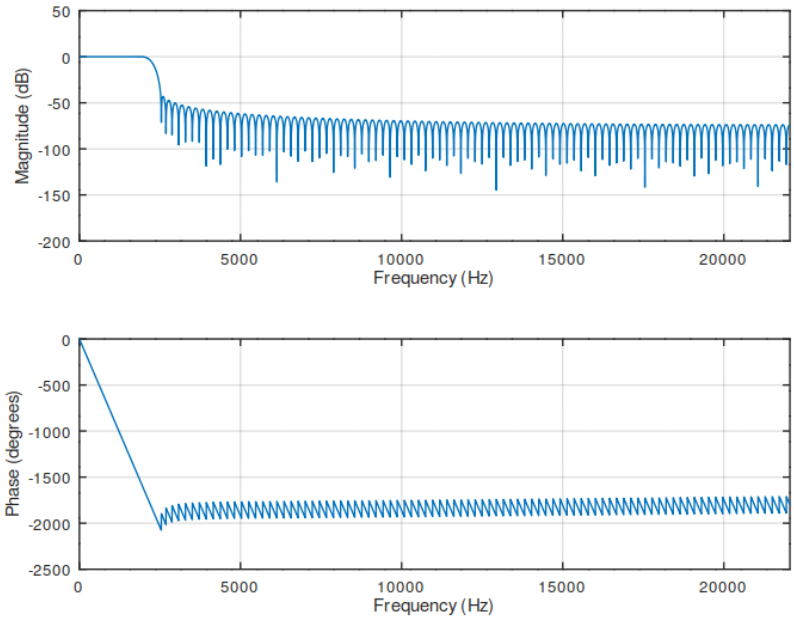


7. Ábra: Konvolúció másodfokú szűrőre

Visszacsatolás nincsen a FIR szűrőben, mindig csak a bemenet aktuális és az előtti értékeit használja fel. Egyszerűségéből adódóan gyorsan és könnyen lehet számolni, a számítási igény és a szűrő átmenetének élessége a szűrő fokszámával arányosan nő, így egy optimalizációs problémát vet fel: el kell dönteni, hogy mi az a fokszám amin a számításokból adódó erőforrásigény még vállalható az alkalmazás számára, illetve a célnak kellően pontos. Stabilitás szempontjából a kimenet mindig stabil, állandó késleltetéssel minden frekvencián, lineáris fázisfüggvény igénye esetén használjuk főként. Másik neve a konvolúciós szűrő.

A koefficienseket Octave segítségével állítom elő, a mintavételi frekvenciát az USB-s hangkártyához igazítom. Példa egy kétszázad rendű FIR szűrő előállítására, ami 2 kHz frekvenciáig engedi át a jelet és 2,5 kHz frekvenciánál van a zárósávja:

```
fs=44100;
b=firls(200-1, [0 2000 2500 fs/2]/(fs/2),[1 1 0 0]);
freqz(b,1,1e4,fs)
```



8. Ábra: Kétszázad fokú FIR szűrő megvalósítása Octave segítségével.

5.2 Implementációk C nyelven

Készítettem egy programkeretet, amely a különböző implementációk közötti váltásokat támogatja. Szeretném megvizsgálni az egyes megoldásokat többféle lebegőpontos számábrázolással, viszont C nyelven nehéz generikus kódot írni. Ezt még az eszközök különbözősége is nehezíti. Erre megoldást az nyújt, hogy nem akarok egyszerre egy időben kétféle mérést is elvégezni ugyanazon az eszközön, ezért különböző makrókkal kezelem a felmerülő nehézségeket:

1. A processzor úgy tud leghatékonyabban számokkal dolgozni, hogy az operandus mérete megegyezik a regiszter méretével: így nem kell se előjellel kiegészíteni a kisebb méretű számot, se leválasztani biteket a nagyobból. Mivel a Pi 3B+ már 64 bites, ezért ott külön 64 bites változókat lenne célszerű használni, a többi eszköznél 32 biteset. Definiáltam egy saját előjeles és pozitív egész típust aminek méretét egy makróval be lehet állítani fordítási időben.
2. A 32 és 16 bites lebegőpontos való mérések elkülönítésére is egy makrót használok. Definiáltam a szűrőegyütthatóknak egy saját típust, ami a makró

értékétől függően típusa az alapesetben rendelkezésre álló lebegőpontos, vagy 16 bites lebegőpontos. A szűrők teszteléséhez használt paraméterek típusát is ezen makró segítségével állítom be.

3. A nyomkövetés fontos a programtervezésnél, viszont a méréseket jelentősen lassítani tudja. Láthattuk már, hogy egy egyszerű sztenderd kimenetre való kiírás mennyire sok utasításból áll valójában, a „printf” is hasonlóan, rendszerhívásokkal működik. A kiíratások makró állapotától függetlenül megjelennek a kódban, viszont a makrók felülírják viselkedésüket. Engedélyezve úgy működnek, ahogy kell nekik, viszont tiltott esetben az előfeldolgozó üresen hagyja a helyüket. A kiíratáshoz szükséges fejlécet is csak engedélyezett állapotban illesztem be.
4. Az éppen aktuális implementációt és fokszámát is a makró értékével választhatjuk ki.

Egyszerűség kedvéért a szűrőnek a változói globális memóriaterületen vannak jelen, így az összes implementáció összes függvénye el tudja érni.[2] A tesztek során a globális változók nem interferálhatnának a teszt függvényein kívül semmivel, viszont a helyes gyakorlatért statikus változóknak definiálom őket, így a fájlon kívül máshonnan nem lehet elérni őket.

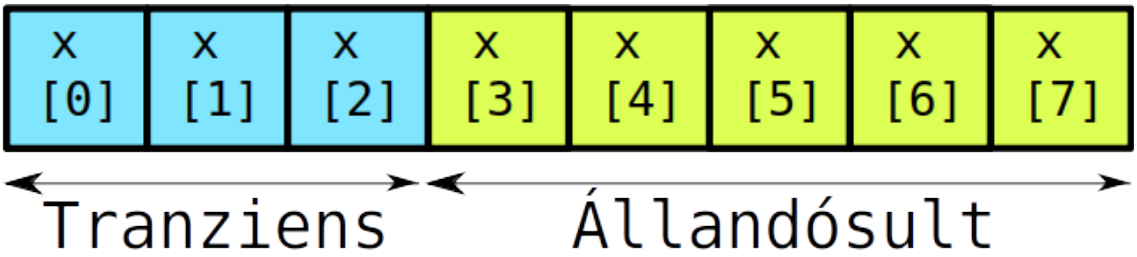
A továbbiakban bemutatok néhány lehetséges implementációt, amivel később a feldolgozást el lehetne végezni. Az implementációk mérései alapján választom ki azokat, amiket a jelfeldolgozásnál használok.

5.2.1 Implementáció definíció alapján

A továbbiakban erre az implementációra „hagyományos” vagy „definíció alapú” megvalósításként hivatkozok. Ebben az implementációban nincsenek szoftveresen alkalmazott vektor utasítások, a vektorizálás a fordítóra van bízva teljesen, ellentétben a Neon alapú implementációkkal. Konvolúció során a műveletek egyesével történnek, nincsenek szoftveres vektorműveletek.

[2]Teszteléshez kényelmes ez az adatszerkezet, viszont jelfeldolgozáshoz pazarló. Tényleges alkalmazás során érdemes áttérni kisebb méretű kimeneti tömbre amennyiben nem állna rendelkezésre elég memória és ezen tömb legrégebbi elemeit modulo címzéssel folyamatosan felülírni, az újakat pedig kiadni a hangkártyának.

Még mielőtt elérjük a kimeneti indexszel a szűrő fokszámát, addig kevesebb tagunk lesz, mint ahányad rendű a szűrőnk. Ez problémát jelent a programban, mivel mindig az előző szűrőfokszámnyi mintát akarjuk visszamenőleg olvasni, ezért lennének olyan olvasások, amelyek érvénytelen memóriaterületről próbálnának meg adatot kinyerni. Egy feltételes értékadással küszöbölöm ki az ilyen olvasásokat még a ciklus előtt, így elég plusz egy elágazást belevinni a programszerkezetbe, viszont ez jelentősen csökkentheti a feldolgozás sebességét. Nagyobb mintaszámok esetén célszerűbb lehet az állandósult állapot határáról indítani a mérést nulla értékű tranziens adatokkal. Az éles feldolgozás során ezzel nem kell foglalkoznom, megfelelő adatszervezéssel áthidalható ez a feltétel.



9. Ábra: Tranziens és állandósult értékek a számítások során

5.2.2 Definíció alapján elágazások nélkül

A konvolúciós függvény működése az előző esethez hasonló, csupán eggyel több paramétert vár. Ezzel szemben az őt becsomagoló függvényben kettő különböző ciklus van: külön ciklus van addig, ameddig a kimeneti index nem éri el a szűrő fokszámát (tranziens) és külön ciklus van a már állandósult állapotra. Így elkerülhető az elágazások folyamatos kiértékelése, cserébe eggyel több paramétert kell a veremben átadni a függvénynek minden egyes iterációban.

Az új paraméter az első ciklusban megegyezik a ciklus első paraméterével, vagyis az éppen aktuális kimeneti index értékével., második ciklusban konstans értéket vesz fel, a szűrő fokszámával eggyel kisebb értéket. A konvolúciós függvény addig fut, ameddig egyenlő nem lesz a másodiknak kapott paraméterrel.

Kisebb fokszámok esetén lehetőség van kihasználni a fordítónak a „loop unroll” optimalizációs funkcióját (*-funroll-loops*), ami kisebb iterációs számoknál kibontja a ciklus tartalmát, ezzel is csökkentve a ciklusvégi elágazások számát. Így kisebb számok

esetében lehetséges, hogy gyorsabban fut le az így megnövelt kódunk. Nagyobb számok esetén is lehetőség van kibontani a ciklusokat, viszont általában ez a program lassítását eredményezi. [27]

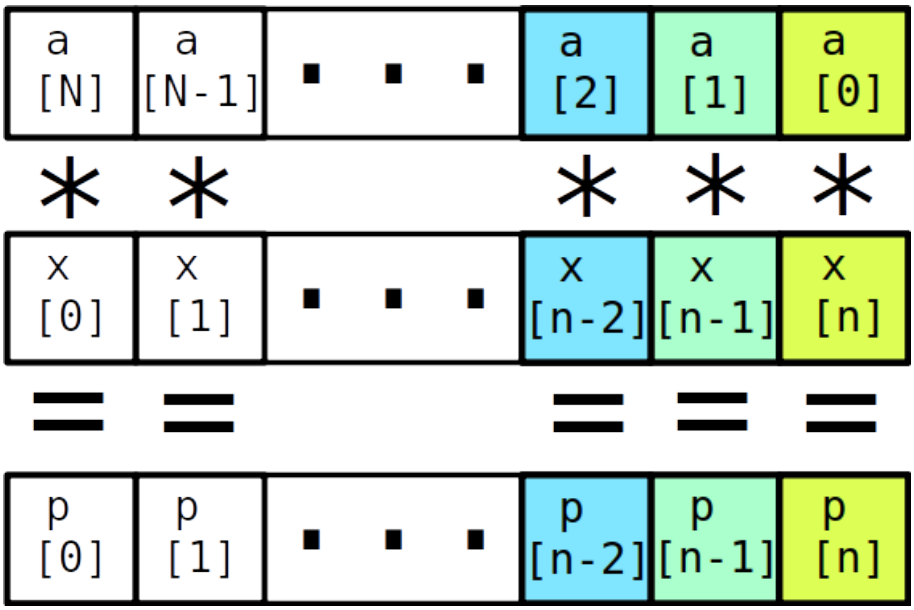
5.2.3 Neon

A Pi 3B+ által használt 64 bites architektúra miatt rendelkezésünkre állnak 128 bites vektorok. Az alkalmazástól és a szűrőnk fokszámától függően kell felosztanunk ezt a 128 bitet: használhatnánk 16 darab 8 bites sávot, de ez a felbontást 8 bitesre szűkíti. 16 bites adatokat használva használhatnánk egyszerre nyolc különböző sávot, tehát legfeljebb nyolcadrendű szűrőt könnyedén tudnánk kezelni a lehető legegyszerűbben. Az algoritmusok vektorának ezért 16 bites előjeles egész számokat vagy 32 bites lebegőpontos számokat érdemes választani.

Az algoritmust felosztom három részre: Az első részben definíció alapján elvégzem az algoritmust, ameddig nincsen legalább annyi mintánk, mint amennyi a szűrőnk fokszáma. A második részben, amikor már van legalább szűrőfokszámnyi mintánk, vektorok segítségével végzem el a műveleteket. Harmadik részben, az utolsó pár darab mintán végzem el a számításokat, amiket már nem lehetne vektorokkal elvégezni.

A második részben felveszek egy koefficiens és bemeneti vektort amin a műveleteket végezni fogom. A bemeneti vektorba a feldolgozási ciklusban töltöm be az adatokat, mindig annyit amennyi éppen belefér, a feldolgozási ciklusban a vektort mindig eggyel léptetem. A betöltés után konvolválom az adatokat a koefficiens vektorral, majd az eredményt vektorként adom vissza a kimeneti tömbnek.

Abban az esetben, amikor a szűrőnk fokszáma legfeljebb annyi, mint a sávszámunk, a konvolúció egyszerűen megvalósítható. Feltételezzük, hogy a koefficiensek fordítási időben rendelkezésünkre állnak és konstansak, valamint a vektorban az egyes értékek „visszafele” helyezkednek el a sávokban. Ez azért szükséges, mert két vektort szorozhatunk sávonként, viszont keresztbe nem. Beolvassuk a bemeneti vektort az éppen aktuális bemeneti tömb mutatója alapján, majd az éppen beolvasott vektor elemeit összeszorozzuk a koefficiensek vektorával.



10. Ábra: Konvolúció megvalósítása vektorokkal

Ezután már csak összegeznünk kell az eredményként kapott „p” vektor sávjaiban található értékeket. Erre kétféle lehetőség is van: tömbökhöz hasonlóan egy ciklus segítségével egyesével hozzáadjuk az eredményt tároló vektor sávjaiban lévő értékeket, vagy kihasználjuk a rendelkezésre álló vektorösszeadásokat. A 16 bites értékeket páronként összeadhatjuk egy négysávos 32 bites vektorba, majd a műveletet megismételve kaphatunk egy kétsávos 64 bites vektort. A 64 bites vektor elemeit összeadva, majd 16 bitessé konvertálva megkaphatjuk az eredményt. Mivel a páronkénti összeadáshoz nincsen lebegőpontos adatokat támogató intrinsics, ezért kénytelenek vagyunk ilyen esetben a vektor sávjain egyesével végighaladni és összegezni az értékeket. Pszeudokóddal leírva a folyamat második részét a fentebb leírt esetet feltételezve:

```
Ciklus amíg a Ciklusindex != Mintaszám

    Bemeneti_vektor = Betölt(Bemeneti_tömb_mutató)
    Eredmény_vektor = Szorzás(Bemeneti_vektor, Koeff_vektor)
    Kimeneti_tömb[Ciklusindex] = Összegez(Eredmény_vektor)
    Ciklusindex++
```

5.3 Implementációk Rust nyelven

5.3.1 Definíció alapján

Az algoritmus megegyezik a C nyelv esetén bemutatottal. Adatszerkezetbeli különbségek adódnak: a kimeneti tömb a „main” függvény része, mivel statikus globális változók módosítása Rust esetén nem biztonságos.

5.3.2 SIMD

A C-s Neon implementációkhoz hasonlóan vektorokat használok itt is. A koefficienseket ismertnek tekintem fordítási időben, ezért elhelyezhetem fordított sorrendben őket a memóriában. A megvalósítás elve a következő: a koefficiens adatszerkezetét megtöltöm mindig annyi elemmel, amennyi belefér. Ha nem elég, akkor több ciklusban kell elvégezni a kimeneti az egy időponthoz tartozó kimeneti érték számítását. SIMD esetén tömbök szeleteivel („slice”, ami a tömbnek egy kisebb, intervallummal címezhető része) lehet inicializálni az adatszerkezetet. A bemeneti vektort feltöltöm a „koefficiens helyzetével párhuzamosan”, vagyis az összeszorzandó elemek egy sávban lesznek. A kettő vektort összeszorzom, majd a szorzat sávjain végighaladva összegzem az értékeket.

A megoldás hátránya, hogy a biztonságos memóriaelérés miatt mindig annyi sávnyi nullával kell feltölteni a bemeneti tömböt, mint amennyi a szűrőfokszám és a sávhosszúság hányadosának felkerekített része. Ez a feldolgozásban mindig valamennyi sávnyi késleltetést jelent, viszont a teljesítménymérésben ugyanannyi elem lesz feldolgozva a méréseim során, csak a bemeneti tömb végéről az elejére kerülnek az elemek amik amúgy is felhasználásra kerülnének. Ez FIR szűrő esetén magasabb fokszámoknál elviselhetetlen, ezért a definíció alapú implementáció kombinálásával itt is ki lehet küszöbölni ezt a problémát. Pszeudokódja megegyezik a C nyelven implementált FIR szűrő Neon pszeudokódjával, csak a betöltő és műveletvégző függvények működése tér el.

5.3.3 Neon

Jelenleg kísérleti állapotban van a Neon támogatás Rust-nál, ezért az intrinsics-ek jelentős része nem támogatott[3]. Ebből kifolyólag a Neon implementációt csak egy speciális esetre valósítottam meg: 4 sávos 32 bites lebegőpontos vektorokra.

Az algoritmus megegyezik a SIMD esetén alkalmazottal, viszont a vektorbetöltést és a műveleteket más utasítások végzik el. A vektorbetöltésért felelős intrinsics sem támogatott jelenleg, ezért a vektor struktúráját egyesével kell kód szinten feltölteni. A szorzás művelet sem támogatott, ezért a szorzatot egy új vektorként mentem el, amelynek sávjaiban egyesével adom meg a tényezőket. A vektor sávjainak összegzése sem lehetséges ciklusban, mert csak fordítási időben ismert értékkel lehet címezni a vektort. Ez nem is feltétlenül baj, a ciklusok kibontásának lehet jótékony hatása is a sebességre alacsonyabb ciklusszámoknál („loop unroll”).

Létezik egy olyan utasítás, ami a szorzást és összeadást valósítja meg, viszont jelenleg csak kétsávos 16 bites előjeles egész számokra támogatott az utasítás.

[3] *core_arch* crate, használva: 2019. november 1.

6 IIR szűrők

6.1 Általános ismertető

Az ismertetőt a MIT OpenCourseWare [25] leírása és Méréselmélet tankönyv [26] alapján készítettem el. Az Infinite Impulse Response (IIR) szűrő impulzusválasza exponenciálisan lecsengő, így el tud érni egy szintet, ahol már a zajjal összehasonlítható az értéke, de nem véges a FIR-rel ellentétben. Az IIR szűrő két részből áll: egy autoregressziós részből (AR) és egy mozgó átlagból (MA). Előbbi tag a FIR tagot írja le, utóbbi ehhez egy rekurzív visszacsatolást ad. Az IIR szűrőt a következő egyenlet írja le:

$$y(n)=\frac{1}{a_0}\left(\sum_{k=0}^{N} b(k)\,x(n-k)-\sum_{j=1}^{P} a(j)\,y(n-j)\right) \tag{2}$$

- y(n) a kimeneti diszkrét értelmezési tartományú szűrt jel
- x(n-k) a bemeneti diszkrét értelmezési tartományú jel az „n-k” időpillanatban
- b(k) bemeneti jel együtthatói (visszafele ható szűrőegyütthatók)
- a(j) szűrt válasz együtthatói (előreható szűrőegyütthatók)
- a_0: az aktuális válasz együtthatója, általában normalizálva van
- N, P a szűrő fokszáma (együtthatók száma).

Másodfokú tag esetén a képlet a következő alakú:

$$y(n)=a(0)\,y(n)+a(1)\,y(n-1)+a(2)\,y(n-2)+b(0)x(n)+b(1)x(n-1)+b(2)x(n-2) \tag{3}$$

$$\begin{aligned} y(0)&=a(0)\,y(0)+b(0)x(0)\\ y(1)&=a(0)\,y(1)+a(1)\,y(0)+b(0)x(1)+b(1)x(0)\\ y(2)&=a(0)\,y(2)+a(1)\,y(1)+a(2)\,y(0)+b(0)x(2)+b(1)x(1)+b(2)x(0) \end{aligned}$$

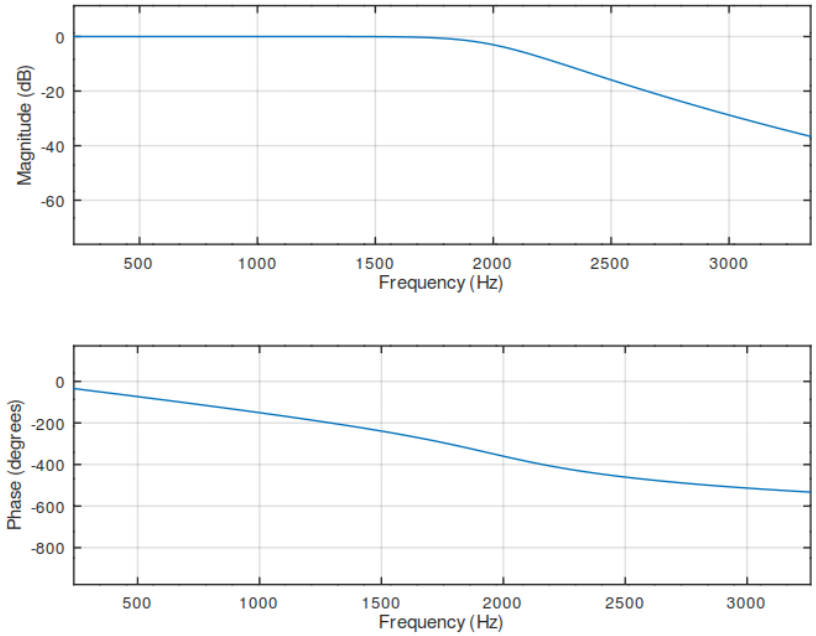
Az IIR szűrő nagy előnye, hogy elég kevesebb fokszám ugyanazért a hatásért amit a FIR szűrő létre tud hozni, ezért működése gyorsabb. Hátrányai, hogy a kimeneti értékeket el kell tárolnunk, és sokkal érzékenyebb a kvantálásra, ezért akár instabil is lehet. Ezek ellenére a real-time szűrők általában IIR szűrők, amennyiben lehetséges stabilan implementálni.

Elterjedt nevük még a „rekurzív szűrők”. Néhány IIR szűrőtípus röviden:

- Butterworth: Amplitúdófüggvénye maximálisan lapos az átviteli- és zárósávjában egyenletesen csökken, viszont széles átmenete van. Előnyük a simaságuk és frekvenciamenetük.
- Inverz Chebysev: Meredekebb az átmenet, mint a Butterworth szűrő, viszont hullámos a zárósávjában.
- Chebyssev: Hullámos az átviteli sávban, viszont élesebb, mint az Inverz Chebysev ugyanazon a fokszámon.
- Bessel: Széles átmenete van, maximálisan lapos amplitúdó- és fázisfüggvénnyel. Utóbbi közel lineáris az átviteli sávban.
- Cauer: Hullámos mind az átviteli-, mind a zárósávban, viszont a legmeredekebb az említettek közül.

Teszteléshez a szűrőegyüttthatókat Octave segítségével állítom elő. A „b” vektorban a bemenethez, az „a” vektorban a kimenethez tartozó szűrőegyüttthatók találhatóak meg. A mintavételi frekvenciát itt is a hangkártyák hardveréhez igazítom.

```
fs=44100;
[b,a]=butter(8, 2e3/(fs/2));
freqz(b,a,1e4,fs)
```



11. Ábra: IIR szűrő megvalósítása Octave segítségével

6.2 Implementációk C nyelven

Az előző fejezethez hasonlóan a következőekben bemutatok néhány lehetséges implementációt, amiket használhatok a jelfeldolgozás során.

6.2.1 Implementáció definíció alapján

A továbbiakban erre az implementációra „hagyományos” vagy „definíció alapú” megvalósításként hivatkozok. Megvalósítása hasonlít az előző fejezetben megvalósított FIR szűrőkre. Itt egy kimeneti érték előállításához kettő konvolúciót kell elvégezni: egyet a bemeneti adatokra, egyet a válaszra visszamenőlegesen, majd ezeknek az összegét leosztani az éppen aktuális válaszhoz tartozó szűrőegyüttható val. Utóbbi művelet, az osztás, új problémát vet fel egész számokon végzett műveletek esetén amennyiben ez az együttható nem egyezik meg eggyel. Mivel a Pi rendelkezik FPU-val, ezért ettől a problémától eltekintek. Pszeudokóddal leírva egy IIR szűrő fixpontos törtszámábrázolással:

```
Ciklus amíg a Ciklusindex != Mintaszám
```

```
Bemeneti_tag = Konvolúció(Bemeneti_tömb_mutató, KoeffB, index)
Kimeneti_tag = Konvolúció(Kimeneti_tömb_mutató, KoeffA, index)

Kimeneti_tömb[Ciklusindex] = (Bemeneti_tag - Kimeneti_tag)/A0)
Ciklusindex++
```

6.2.2 Neon

A C nyelven megvalósított FIR szűrőhöz hasonlóan működik ez a Neon implementáció is. A szűrőegyütthatókat fordított sorrendben betöltöm külön vektorokba, amiknek értékeit ciklus segítségével töltöm be minden egyes kimeneti értékre. Ezek után betöltöm a kimenet és a válasz vektorát is külön-külön, amiket összeszorzok a nekik megfelelő koefficiens vektorral. Mivel megcseréltük a koefficiensek sorrendjét és mindig ugyanazon elemmel szeretnénk a különbséget leosztani, ami ezen kívül nem szabad, hogy szerepeljen a szorzó-akkumulációban, ezért a kimenethez tartozó koefficienseket el kell tolni a memóriában egy lépéssel. A tömb elejét egy nullával kiegészítve a várt működést kapjuk tetszőleges szűrőfokszámra.

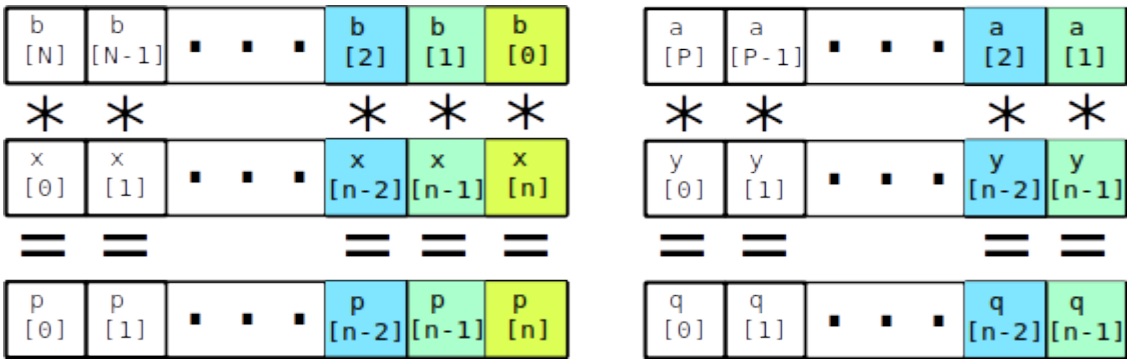
```
Ciklus amíg a Ciklusindex != Mintaszám
     sumX = 0
     sumY = 0
     Ciklus amíg van Koefficiens

          KoeffA_vektor = Betölt(KoeffA_tömb_mutató)
          KoeffB_vektor = Betölt(KoeffB_tömb_mutató)

          Bemeneti_vektor = Betölt(Bemeneti_tömb_mutató)
          Kimeneti_vektor = Betölt(Kimeneti_tömb_mutató)

          Bemeneti_vektor = Szorzás(Bemeneti_vektor, KoeffB_vektor)
          Kimeneti_vektor = Szorzás(Kimeneti_vektor, KoeffA_vektor)

          Eredmény_vektor = Kivon(Bemeneti_vektor,  Kimeneti_vektor)
          Ciklusindex += Szűrőfokszám //Vektor sávjainak száma
     Kimeneti_tömb[Ciklusindex + Sávszám -1] =
               Összegez(Eredmény_vektor)
```



12. Ábra: IIR szűrés Neon segítségével

6.3 Implementációk Rust nyelven

6.3.1 Definíció alapján

Az algoritmus itt is megegyezik a C nyelv esetén bemutatottal a kimeneti tömb memóriabeli elhelyezésén kívül.

6.3.2 SIMD

A FIR-es SIMD implementációhoz hasonlóan működik. A használt vektorok adatbetöltését a kimeneti cikluson belül egy másik ciklus szervezi, ami nullától addig megy, amennyi a szűrőfokszám és a sávhossz hányadosának felső egész része.

Az adatok betöltése után másik SIMD adatszerkezetekben tárolom a szorzatvektorokat. SIMD-nek köszönhetően a művelet egyszerűen a szorzás operátorral leírható. Az eredményeket az előzőekhez hasonlóan, sávonként összegzem. Pszeudokódja megegyezik a C nyelven implementált IIR szűrő Neon pszeudokódjával, csak a betöltő és műveletvégző függvények működése tér el itt is.

6.3.3 Neon

Az előző SIMD implantációhoz és a FIR szűrő esetén bemutatott Neon implementációhoz hasonlóan készítettem el ezt a megoldást, egy speciális esetre a modul jelenlegi hiányosságaiból adódóan.

7 Mérési eredmények

Mivel több egymástól lehetségesen függő mérési szempontom van, ezért felveszek egy alapértelmezett környezetet, amitől az egyes méréseknél a szempont függvényében eltérek. Alapértelmezetten 600 MHz frekvencián, gyorsítótár kiürítése nélkül és külső hűtéssel Raspberry Pi 3B+ eszközön. A többi modellen az operációs rendszerrel együtt járó kernelt használom. Iterációk szempontjából minden mérés ezer iterációval fut, iterációkon belül száz ismétlődéssel. Az iterációszám mérések során, tapasztalati úton alakult ki. Elegendően sok mérés ahhoz, hogy kellően sok minta legyen ábrázolható és összehasonlítható, valamint a mérési idő is elviselhető: fél órától egy óráig tart egy ilyen mérés.

A szűrőket illetően, FIR szűrő esetén egy ötvenkét fokút, IIR szűrő esetén ötöd fokút használok kétszáz darab mintával. Alapesetben a C-s implementációkat mérek.

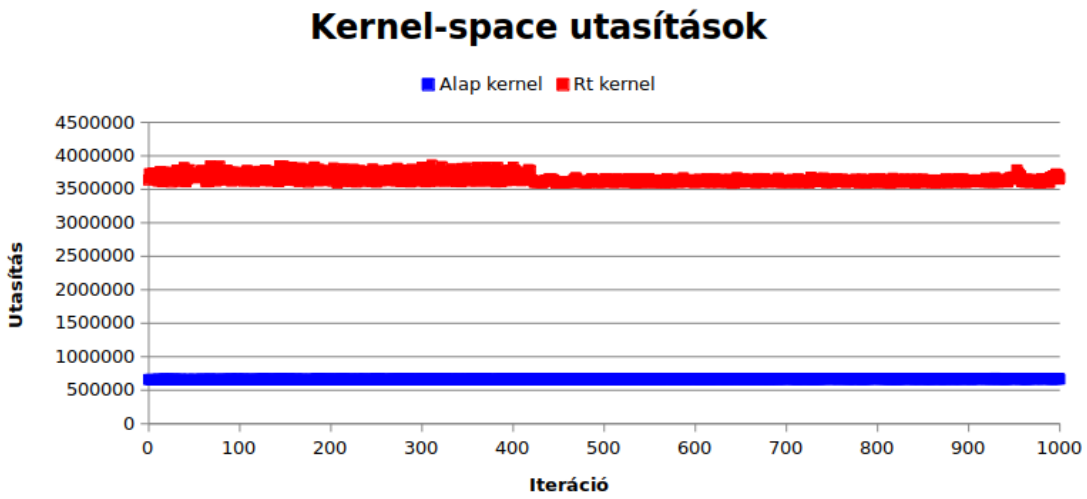
A mérésben elsősorban utasításokat, ciklusokat és futási időt mérek. Utasítások mértékegysége maga az utasítás, ciklusoknál a ciklusok száma, valamint futási időnél milliszekundum, ami egy iterációra vonatkozik. Többszörös ismétlődés esetén az átlagot jelölik.

7.1 Kernel és izolált magok vizsgálata

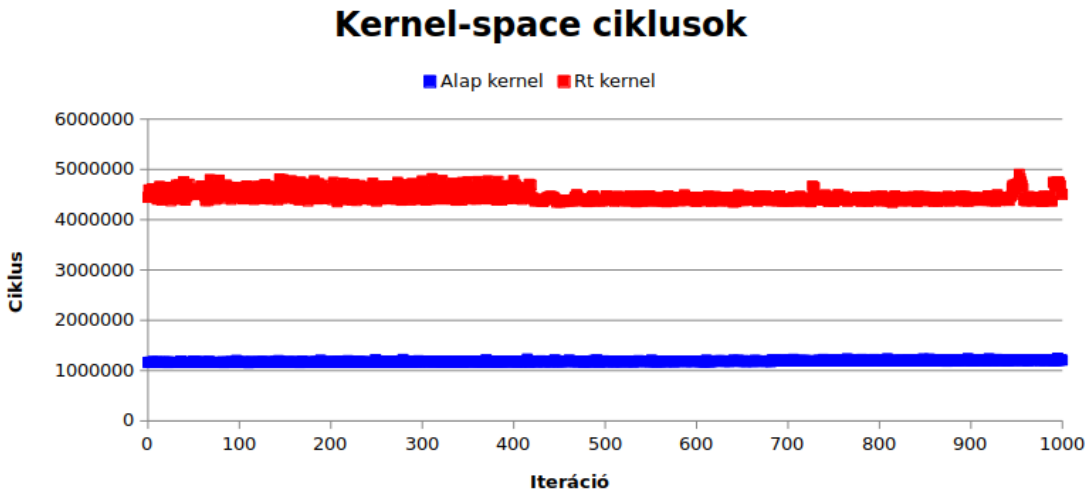
Első mérés eredményeként az alapértelmezett kernelt akarom kijelölni a további méréseimhez. Ehhez a definíció alapú FIR szűrő implementációját mérem kettő különböző kernelen, ami a legegyszerűbb mérés az összes közül. A mérést csak ezer iterációig engedtem, mert félúton már látványosan elkülönült a kettő kernel egymástól.

Általánosságban az általam fordított kernel lassabb. Ez ciklusszámban, utasításszámban és futási időben is észrevehető. „User-space” ciklusokat tekintve stabilan 2,5%-kal lassabb, mint az eredeti kernel. Mivel a kerneleket szeretném összehasonlítani most és nem az implementációt, ezért érdemesebb a „kernel-space” értékeket vizsgálnom. Kernel utasításban stabilan 3,8-4-szerese az eredeti kernelnek, ami miatt már inkább az eredeti kernelt használnám a mérésekhez és a feldolgozáshoz, mint a real-time ütemezővel ellátottat. Utasításokat nézve is 5,5-szerese az eredetinek. Ezeknek oka a túlzott kontextusváltás lehet, ami az eredeti kernel esetén ezer mérés alatt

végig nulla, viszont az új kernel esetén 12-től 40-ig kontextusváltásig előfordulnak értékek. Az ábrákon pirossal láthatjuk az általam fordított kernelt, kékkel az eredetit.



13. Ábra: "Kernel-space" utasítások száma az iterációk függvényében.



14. Ábra: "Kernel-space" ciklusok száma az iterációk függvényében.

Tekintettel arra, hogy a legegyszerűbb esetet mértem jelenleg, nem várok a többi implementációtól drasztikusabb javulást, annak ellenére, hogy optimalizáltabbak lehetnek a definíció alapúnál. Az optimalizáltságuk nem nyilvánulna meg a „kernel-space” ciklusokban és utasításokban.

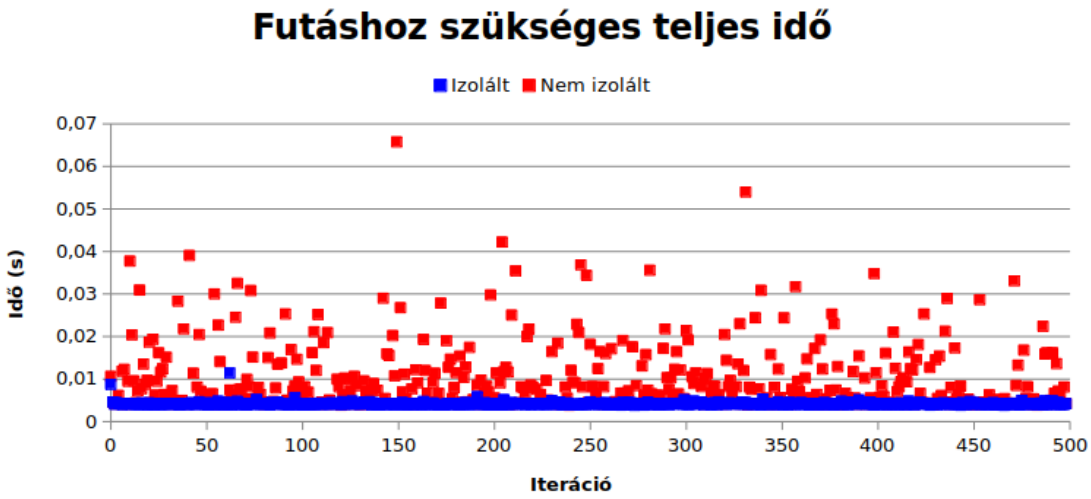
A különbségek egyik oka az lehet, hogy a mérés prioritása rosszul van megadva és a kernel túl gyakran próbál váltogatni a háttérben futó folyamatok és a mérés között. Újabb próbálkozásként a lehető legnagyobb prioritással indítottam el a mérést (legkisebb „nice-value”), viszont a várt eredmény ellentéte jelent meg. A prioritás nélküli eredményekhez képest is lassabb lett.

7.2 Izolált mag és nem izolált mag stressz hatására

A mérés során megvizsgálom, hogy milyen hatással van a mérésekre az, ha szándékosan terheljük a processzort. Izolált mag esetén a terhelés nem kellene, hogy jelentősen látszódjon a méréseken, hiszen a magokon csak a mi „user-space” folyamatunk van. A másik esetben teljesítménycsökkenésre számítok, mivel a mérésnek a stresszel együtt kell futnia. A stresszelést egy egyszerű FIR szűrővel a *stress* csomaggal és annak használati példájának segítségével végzem el.

A futási időbe beleszámít, viszont minden más szempontból megegyeznek az értékek egymással. Például „task-clock” szempontjából nézve mindkét irányban 10 százalékos sávban van a kettő beállítás értékeinek hányadosa.

A perf által mért futási idő viszont a terhelés hatására megváltozik, amit vártunk is a méréstől. Stressz hatására véletlenszerűen szétszóródnak a futási idők amennyiben a stresszelés és a mérés közös magon futnak. Az izolált esetén láthatjuk, hogy a stressznek nincsen különösebb hatása a mérésnél, szinte „egy vonalban” helyezkednek el a mérési pontok.



15. Ábra: Futási idő változása stressz hatására

7.3 Implementációk összehasonlítása

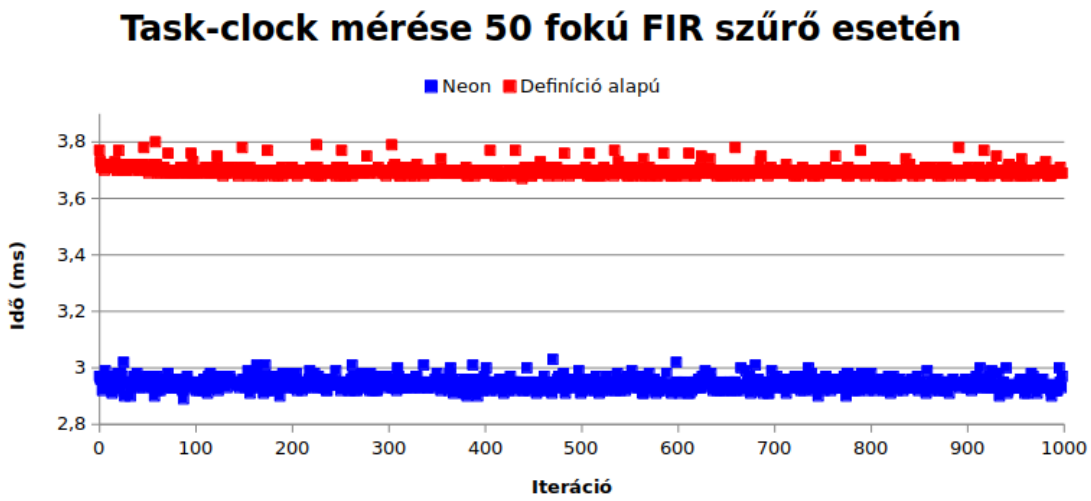
Jelen mérés során összehasonlítom, hogy az egyes implementációk hogyan viszonyulnak teljesítményben egymáshoz.

7.3.1 Implementációk C nyelven

7.3.1.1 FIR szűrők

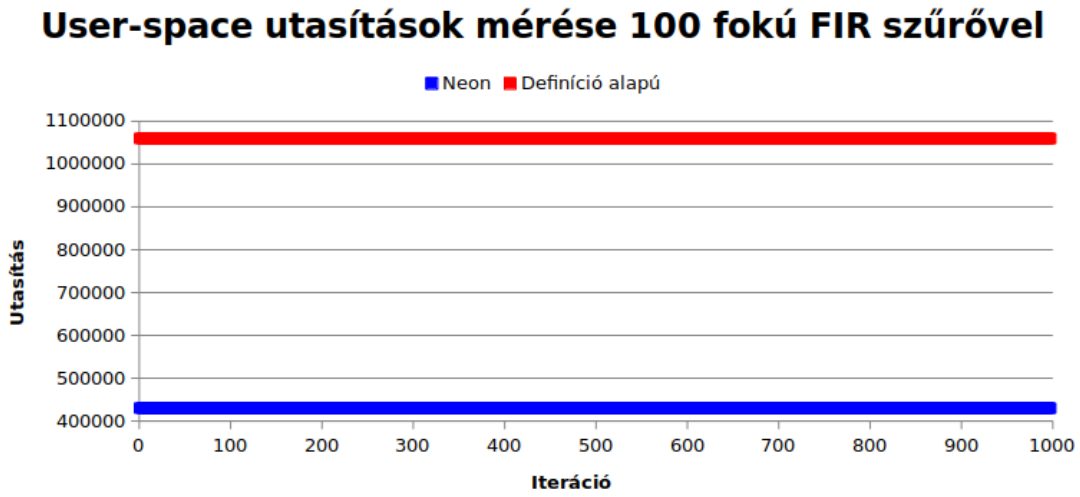
Neon és C implementációk összehasonlítása esetében elmondható, hogy a Neon minden esetben jobban teljesít. Az összehasonlításokat egy később leírt mérés adataival végzem le, 52, 100 és 200 szűrőfokszámra. A jobb teljesítmény alacsonyabb utasításszámot, kevesebb ciklust és futási időt jelent.

Első esetben a ciklusszám 78 százalékkal volt több a definíció alapúnál. Ennél magasabb arányt kapunk utasításszámban, ahol 239 százalékkal több utasítást mértem. A magas számok ellenére a „task-clock” csak nagyjából 25 százalékkal több, mint Neon esetében.



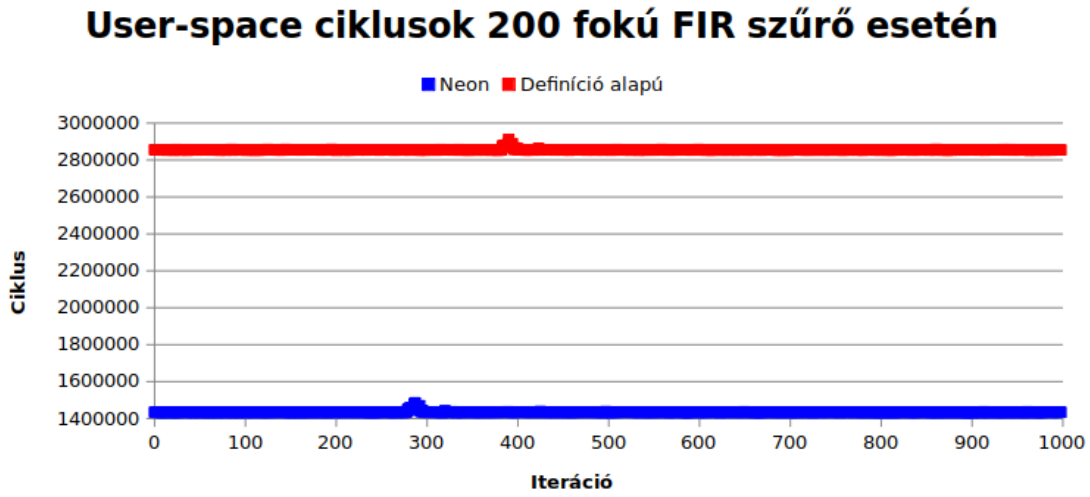
16. Ábra: Task-clock változása az iterációk függvényében Neon és hagyományos implementáció esetén

Második esetben, 100-ad rendű szűrő esetén a ciklusszám esetén 94 százalékkal, utasítások 246 százalékkal volt több, mint Neon esetében. Futási idő tekintetében 40 százalékkal lassabb a definíció alapú.



17. Ábra: Utasítások változása az iterációk függvényében Neon és hagyományos implementáció esetén

Utolsó esetben az utasításszám egészen 99 százalékra növekedett, viszont az utasításszám visszaesett az előző méréshez képest 218 százalékra. Futási időben 56 százalékkal volt lassabb a definíció alapú, mint a Neon.

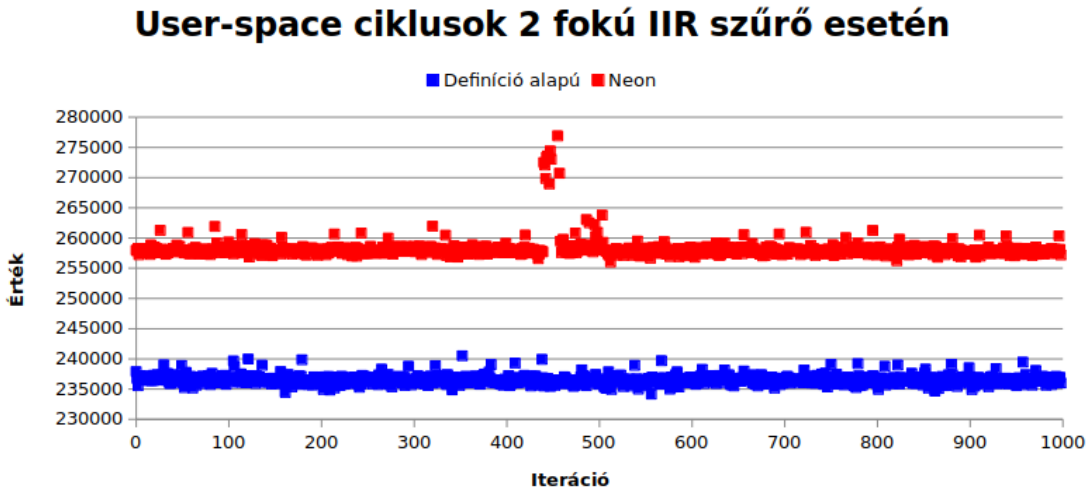


18. Ábra: Ciklusok változása az iterációk függvényében Neon és hagyományos implementáció esetén

Összességében a szűrő fokszámának növekedésével a ciklusszám és a futási idő aránya páronként nő.

7.3.1.2 IIR szűrők

IIR szűrők esetében kettő fokú szűrőre a hagyományos definíció gyorsabbnak bizonyult. Az általam elkészített Neon implementáció 8 százalékkal több ciklust tartalmazott, utasításokból 10 százalékkal több. Task-clock szempontjából ez 1-2 százalékban nyilvánul csak meg. Ez érthető, hiszen a vektor sávszáma több volt, mint a koefficiensek száma, tehát nem éri meg vektorizálni a folyamatot.



19. Ábra: Ciklusok változása az iteráció függvényében másodrendű IIR szűrő esetében

Ötöd fokú szűrők esetén az arányok erősen javultak a Neon implementáció számára. Neon esetén nagyjából 10 százalékkal kevesebb ciklus szükséges a szűréshez, valamint 30 százalékkal kevesebb utasításból áll. Task-clock szempontjából csak 2 százalékkal gyorsabb a Neon implementáció.

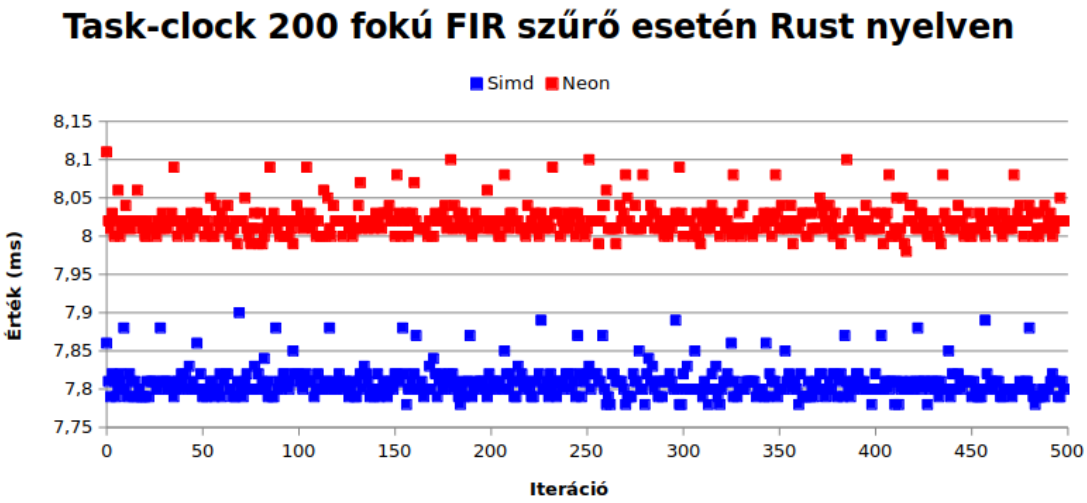
Nyolcadrendű szűrő esetén a kettő implementáció közötti különbségek nőttek. Neon 23 százalékkal kevesebb ciklusból, 48 százalékkal kevesebb utasításból áll. Task-clock aránya 5 százalékra nőtt.

7.3.2 Implementációk Rust nyelven

A mérésé során a SIMD implementációt hasonlítom össze a definíció alapúval és a Neon alapúval. A méréseket 500 iterációra végeztem el 100 ismétlődéssel.

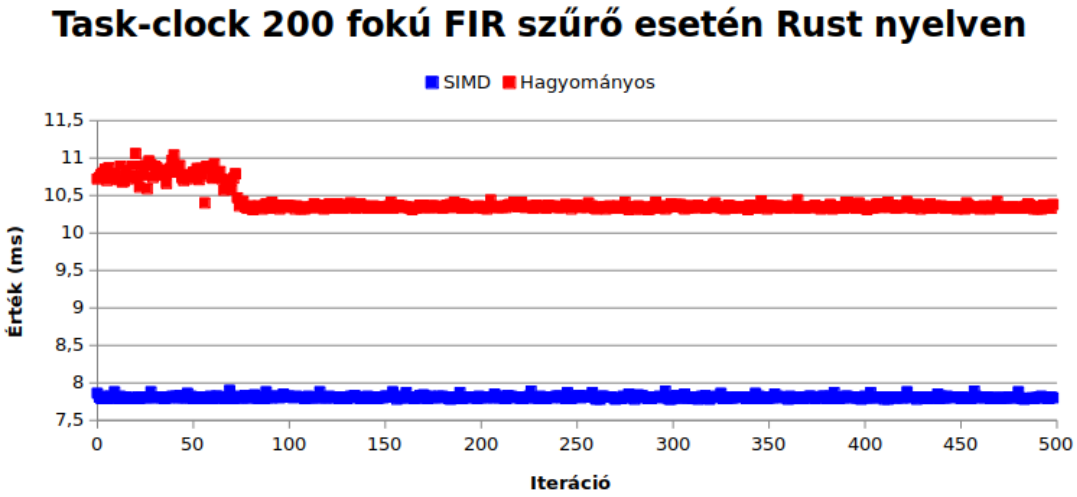
7.3.2.1 FIR szűrők

Általánosságban SIMD technológiával működő szűrőnek kevesebb ciklus és utasítás kell, mint a Neon alapúnak. Ez lehetségesen annak köszönhető, hogy még kezdetleges állapotban van a Neon Rust támogatása. Például 52 fokú szűrő esetén a hagyományos implementáció csak 12 százalékkal tartalmaz több utasítást és 11 százalékkal több ciklust. Ezek ellenére ebben az esetben kevesebb, mint 5 százalékkal több a „task-clock„ ideje. A fokszám növelésével a különbségek csökkenek, viszont még kétszázad rendű szűrő esetén is gyorsabb a SIMD, még ha csak 2 százalékkal is.



20. Ábra: Task-clock változása az iterációk függvényében 200-ad rendű FIR szűrő esetén

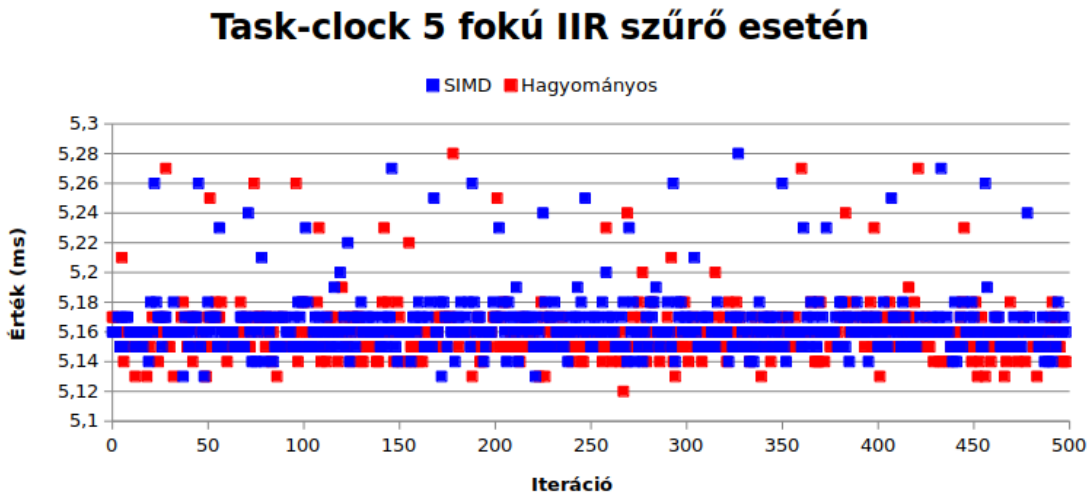
A hagyományos implementációval összehasonlítva a C-s mérésekhez hasonló eredményt vártam. A SIMD technológiával megvalósított szűrő lényegesen kevesebb ciklusból és utasításból áll, mint a hagyományos. Példaként a kétszázad rendű FIR szűrő esetén 36 százalékkal nagyobb a hagyományos implementáció futási ideje, mint a SIMD technológiás. Ez annak is köszönhető, hogy utasításszámban több, mint kétszer annyi kell neki, 67 százalékkal több ciklusból is áll, valamint 40 százalékkal több elágazást tartalmaz.



21. Ábra: Task-clock változása az iterációk függvényében 200-ad rendű FIR szűrő esetén

7.3.2.2 IIR szűrők

IIR szűrők esetén 2 és 5 fokú szűrő esetén a hagyományos implementációnak van a legkevesebb ciklusa és futási ideje. A SIMD technológiás implementációval összehasonlítva task-clock esetén csak 1-2 százalék a különbség, 8 fokú szűrő esetén már a SIMD a gyorsabb viszont. Ciklusokban és utasításokban is 10 százalék alatti a különbség.

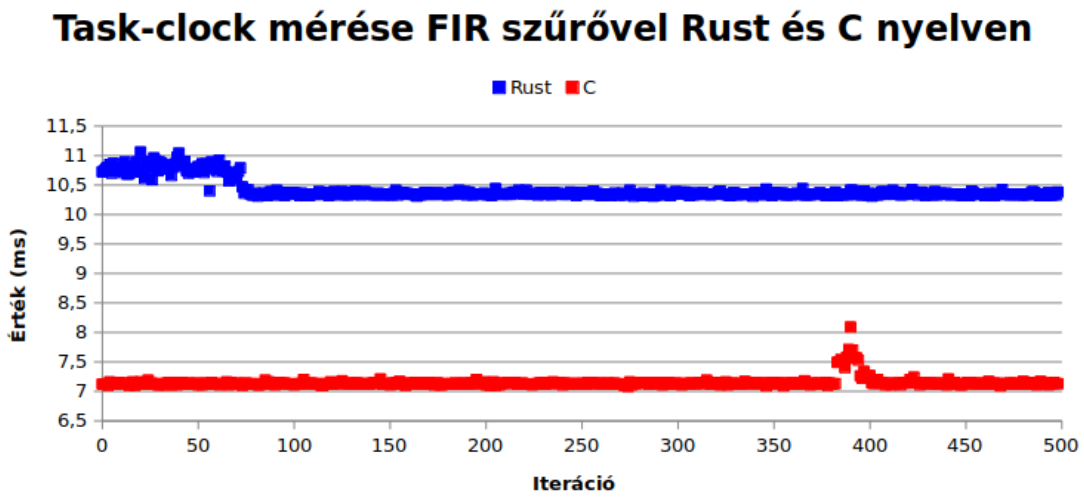


22. Ábra: Task-clock változása Rust nyelven implementált ötödfokú IIR szűrő esetén

SIMD és Neon esetén, SIMD a ciklusok, utasítások és task-clock szempontjából minimálisak a különbségek.

7.3.3 C és Rust összehasonlítása

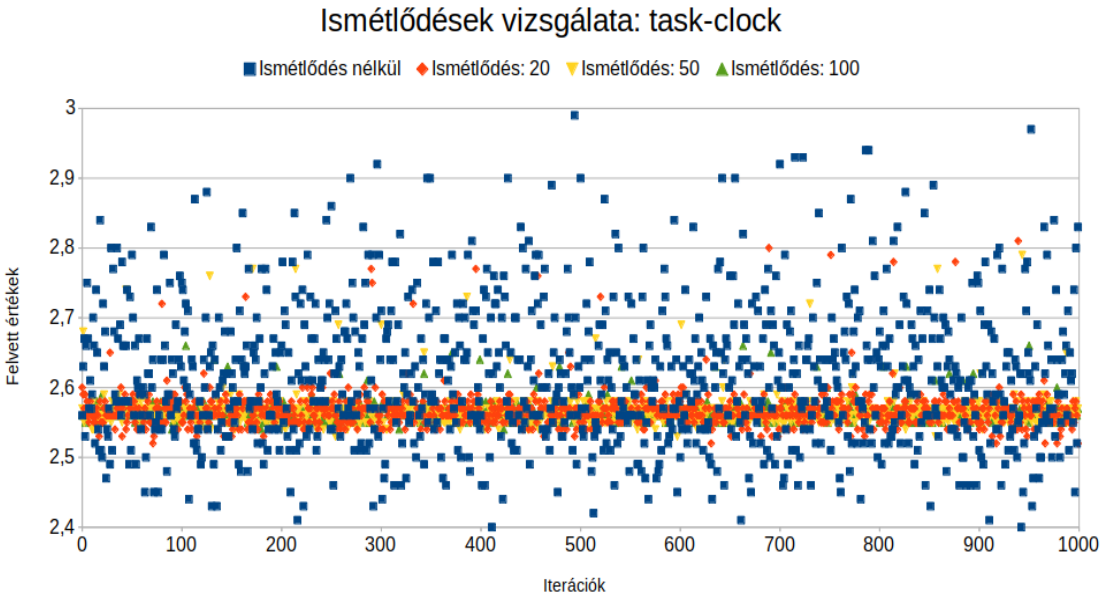
A két nyelv összehasonlítása nem egyszerű feladat, mivel a fordítási paraméterek sok mindent befolyásolnak. Rust esetében az alap fordítás alatt egy „debug buildet” kapunk, ami nincsen optimalizálva. Ennek következtében egy igen lassú állományt kapunk. A „release buildet” már össze lehet hasonlítani a C-s állományokkal. Én FIR esetében a hagyományos implementációkat hasonlítom össze, valamint a C-s Neon és a Rust-os SIMD implementációt. Mindegyikre igaz volt, hogy a C-s implementáció gyorsabb. Ezeknek a mérési eredményeknek vizsgálata és a megfelelő fordítási paraméterek beállítása további kutatást igényel.



23. Ábra: Task-clock vizsgálata hagyományos FIR szűrő esetében C és Rust nyelven

7.4 Iterációk és ismétlődések vizsgálata

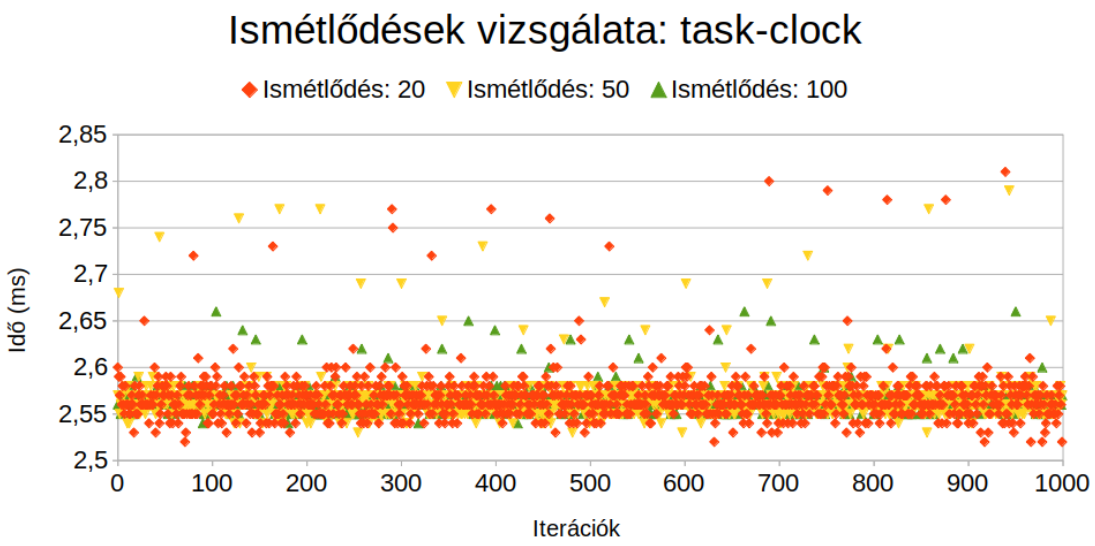
Az iterációk vizsgálata során kiderítem, hogy az ismétlődések nagyszámú jelenléte mennyire különbözik a csak iterációkból álló mérésektől. A definíció alapú FIR szűrőt lemérem 1, 20, 50 és 100 ismétlődéssel 1000 iteráció mellett Pi 3B+ eszközön. A várt eredmény az, hogy minél nagyobb az ismétlődések száma, annál több mérési adatot lehet átlagolni, így ábrázolás során a mérések menete is simább lesz.



24. Ábra: Ismétlődések vizsgálata (task-clock)

Látható, hogy ilyen felbontásban nem lehet megkülönböztetni azokat a méréseket, ahol már vannak ismétlődések. Amennyiben nem ismétlünk iteráción belül, akkor láthatóan a többi méréshez képest hatalmas a felvett értékek szórása. A szórás viszont fontos, ugyanis az egyes elemek futási időket jelölnek, vagyis a kékkel jelölt pontok is valós futási időket jelentenek. Amennyiben ügyelnünk kell feldolgozás során az adatok kiküldésének ütemezésére, úgy a lehető legnagyobb (worst case) értékhez kell igazítanunk az ütemezést.

Amennyiben kivesszük a „kék” mérési pontokat és növeljük a felbontást, láthatjuk a különbséget a 20, 50 és 100 ismétlődéses mérés között. Az értékek ugyanazon főbb sávokban találhatóak meg, csak néhány kiugró értékük van. Minél kisebb az ismétlődések száma, annál több kiugró érték van. Ezen kiugró értékek távolsága a fősávoktól is nő az ismétlődések csökkenésével.

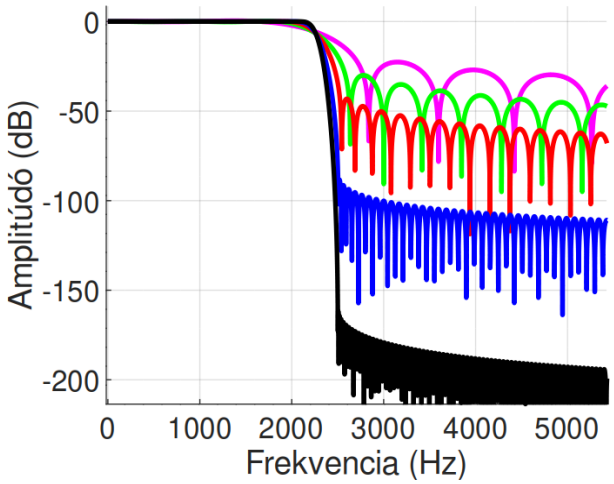


25. Ábra: Ismétlődések vizsgálata (task-clock, 20, 50 és 100)

Minél többször ismétlünk egy mérést, annál jobb lesz az átlagunk és simább a mérési menetünk, viszont a mérési idő jelentősen nő ezzel együtt. Keresnünk kell egy olyan ismétlési számot, ami mellett tudunk méréseket futtatni emberi időben és elfogadhatóan kevés kiugró értékkel.

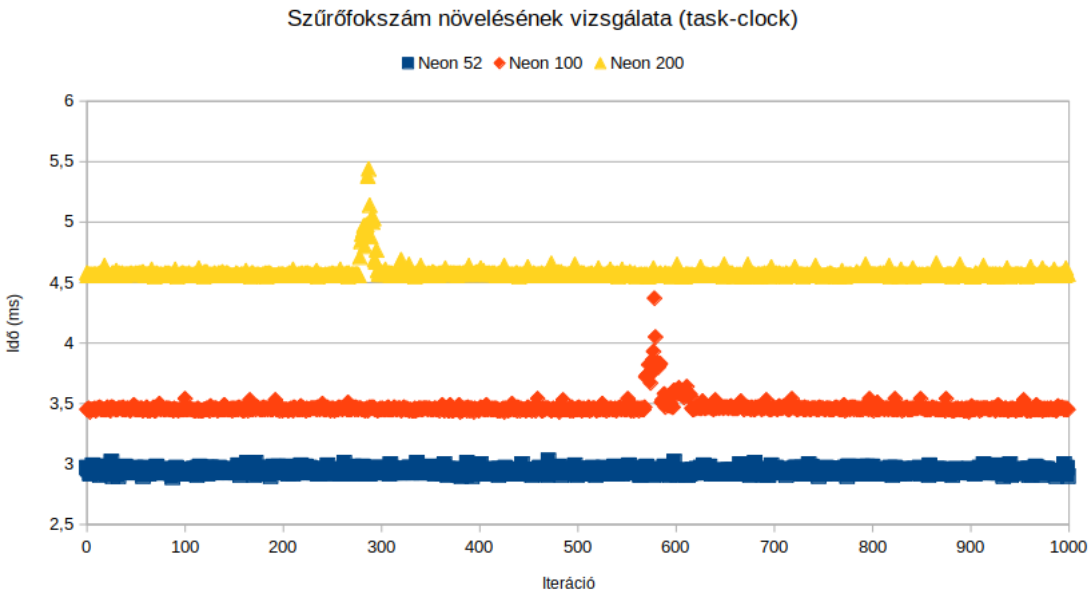
7.5 Mérések különböző fokszámokkal

A szűrő fokszámának növelésével a számítási igény is növekszik, viszont a zárótartományban nagyobb csillapítást kapunk és nagyobb meredekséget az átmeneti tartományban. A következő ábrán rózsaszínnel 50, zölddel 100, pirossal 200, kékkel 500 és feketével 1000 fokszámú szűrő látható, amely 2 kHz-ig átengedi a jelet, 2.5 kHz frekvenciától szűri.



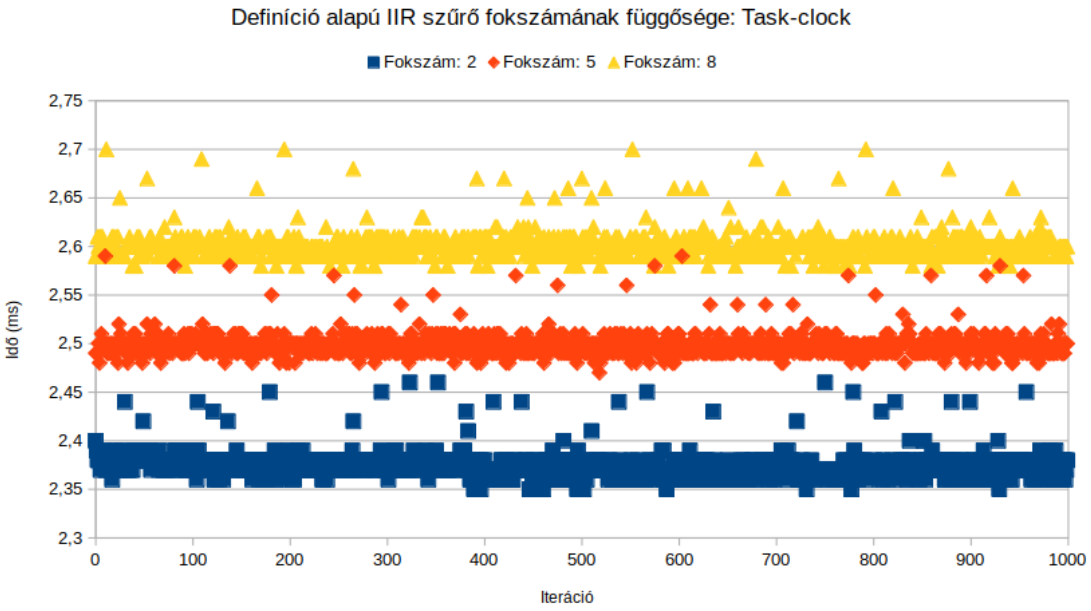
26. Ábra: Különböző fokszámú FIR szűrők csillapításának vizsgálata.

A mérést elvégeztem FIR és IIR szűrőkkel is, Neon alkalmazásával és a hagyományos, definíció alapú implementációkkal is. A mérés során az együtthatók növelésével nőtt az utasításszám, ciklusok száma és a műveletek elvégzéséhez szükséges idő, mint ahogy azt elvártuk. Példaként a FIR Neon implementációk mérésének egyik részeredménye látható. Látható, hogy hiába dupláztuk meg a szűrő fokszámát, az algoritmus elvégzéséhez szükséges idő nem duplázódott. Nem is szabadna, hiszen az általam használt FIR szűrő algoritmusok kezdő és záró része (ahol nem áll rendelkezésre egy teljes vektornyi adat) a definíció alapú implementáció alapján működik, egyedül a közepe használja az adatokat vektorként, így nem is szabadna egyenes arányosan növekednie az együtthatók számával a mérési szempontok eredményeinek.



27. Ábra: "Task-clock" vizsgálata az iterációk függvényében a szűrőfokszámot

Másik példaként a definíció alapú IIR szűrők esete látható. A fokszám növelésével nő a task-clock, viszont az arányossági tényező kicsi. Az utasítások száma sem egyenesen arányosan változik a szűrő fokszámával.

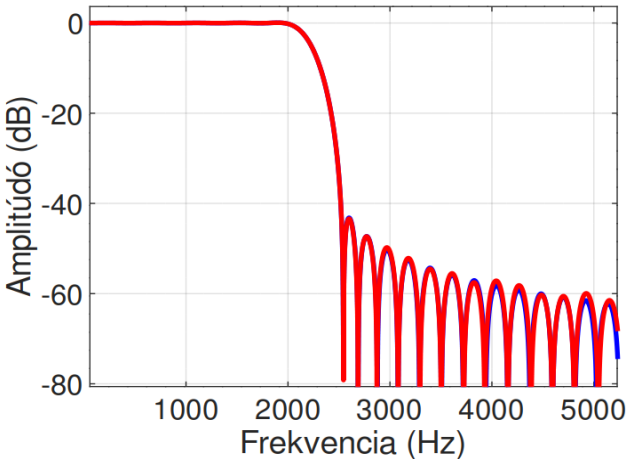


28. Ábra: Definíció alapú IIR szűrő vizsgálata

7.6 Mérések különböző számábrázolásokkal

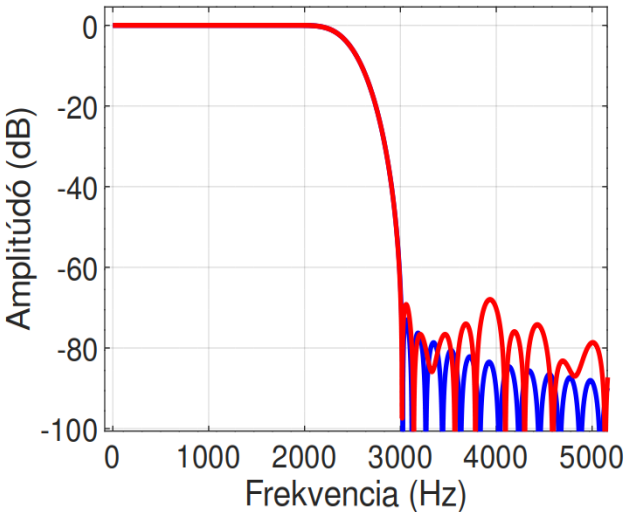
Hardvertől függően különböző számábrázolásokkal különböző sebességgel tudjuk ugyanazt az algoritmust végrehajtani. Ez egy optimalizációs és szűrőtervezési problémát vonhat maga után: beérhetjük rosszabb felbontással és torzult szűrővel a gyorsabb végrehajtásért.

Az általam használt FIR szűrő esetében (200 tap-es, 2 kHz-ig engedi át a jelet, átmeneti sáv 2,5 kHz-ig tart) használhatnék 16, 32 és 64 bites lebegőpontos számokat is, nem lépne fel jelentős torzulás.



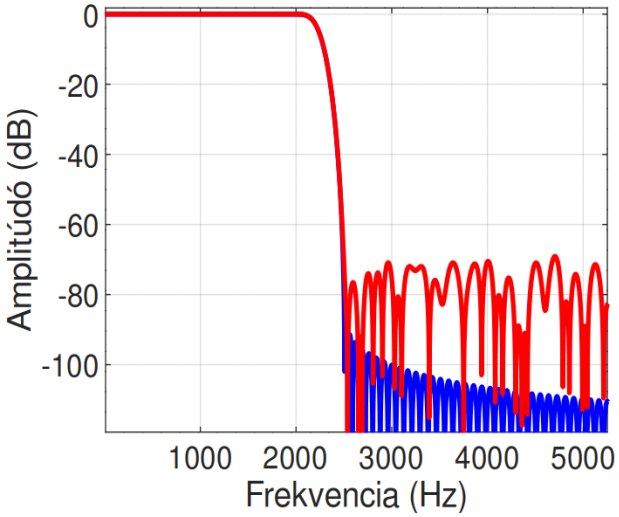
29. Ábra: 64 bites (kék) és 16 bites (piros) FIR szűrő összehasonlítása

Az átmeneti sávot növelve, nagyobb záró tartománybeli elnyomás esetén viszont jelentősebb különbségeket is kaphatunk. Példaként egy 64 és 16 bites 200 tapes FIR szűrő látható, ami 2 kHz-ig engedi át a jelet, átmenet sávja 3 kHz-ig tart. A 16 biten ábrázolt implementációnak látványosan rosszabb a csillapítása a zárósávban.



30. Ábra: 64 bites (kék) és 16 bites (piros) ábrázolású 200 fokszámú FIR szűrő összehasonlítása

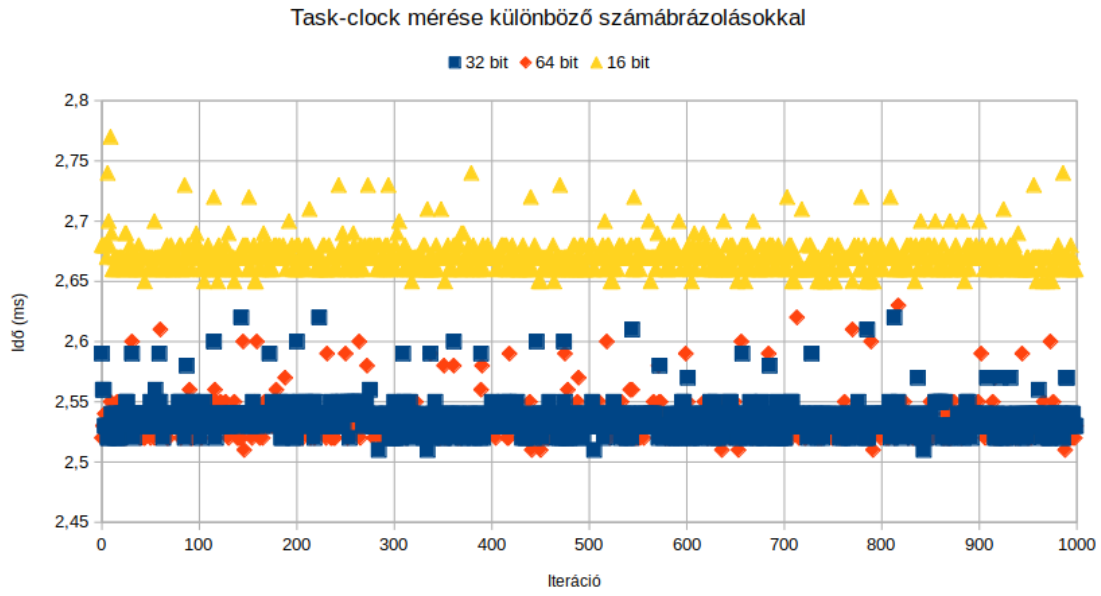
A szűrő fokszámának növelésével is fontosabbá válik a számábrázolás kérdése. Láthattuk a 29. ábrán, hogy egy 200 fokszámú FIR szűrő esetén nyugodtan számolhatnánk akár 16 bites számokkal is, a csillapítást a zárósávban nem befolyásolnál jelentősen. Ezzel szemben már egy 500 fokszámú FIR szűrőnél (további paraméterei megegyeznek a 200 fokszámúval) már ez egyáltalán nem jellemző, szemmel látható a különbség alakul ki a csillapításnál. 32 és 64 bites ábrázolásoknál a különbség még itt sem jelentős.



31. Ábra: 64 bites (kék) és 16 bites (piros) ábrázolású 200 fokszámú FIR szűrő összehasonlítása

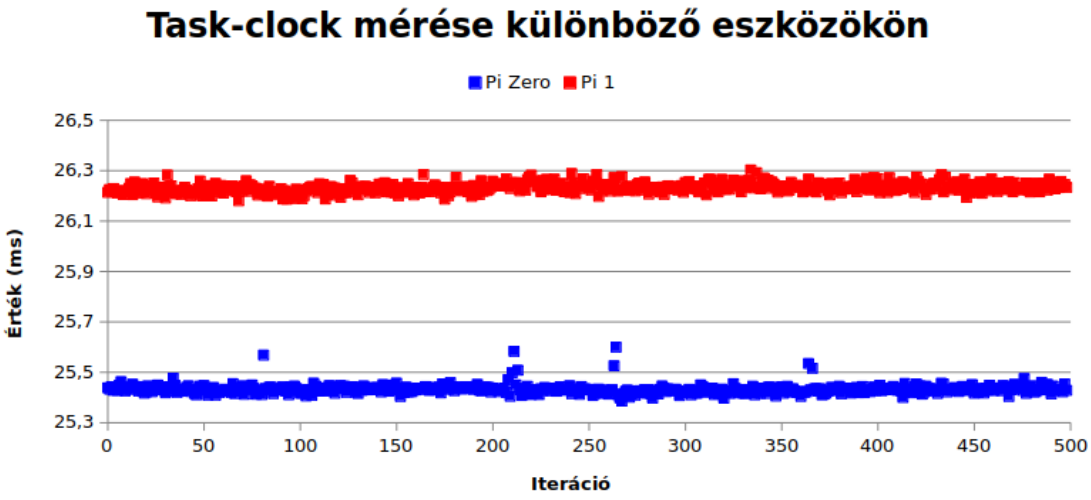
Tehát már a szűrő tervezésekor figyelembe kell vennünk, hogy milyen felbontással fogjuk a feldolgozást elvégezni.

A mérést a definíció alapú FIR szűrővel végeztem el ezer iterációra száz ismétléssel. Alapvető különbséget nem kaptam a 32 és 64 bites mérések vizsgálatakor, fél százalékos tartományban van hányadosuk, viszont a 16 bites mérések ezekhez képest lassabbak voltak 17 százalékkal. Itt is látható, hogy a processzornak könnyebb olyan méretű adatokkal dolgoznia, mint amilyenek a regiszterei. Mivel az IIR szűrő implementációk során ugyanazon építőegységeket használom, ezért elégnek tartom csak FIR szűrővel mérni az ábrázolások által létrejött különbségeket.



32. Ábra: Task-clokk viszgálata Raspberry Pi 3B+ eszközön

Következőként megmérem, hogy a 16 bites átalakítások hogyan teljesít Pi Zero és Pi 1B modellen. Ezeken a modelleken nem képes a perf az utasítások és ciklusok számát mérni. Task-clock-ot viszont képes mérni, ahol nagyjából 3 százalékos különbséget lehet látni. A processzor frekvenciája kötött volt 700 MHz frekvenciára, a mérési beállítások megegyeztek egymással. Az operációs rendszer és a képfájl is megegyezett.

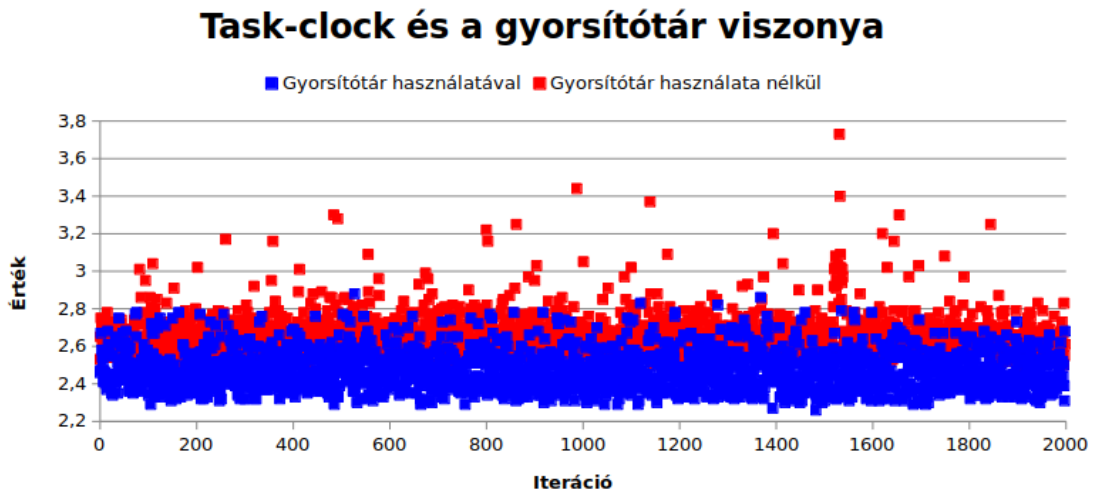


33. Ábra: Pi Zero és Pi 1B modellek FIR szűrése 16 bites lebegőpontos számokkal

7.7 Gyorsítótár hatásának vizsgálata

A mérés célja a gyorsítótár hatásának és a „warm-up” iterációk jelenlétének vizsgálata. A mérés során csak iterációkat használok, nincsenek ismétlődések. Egyik konfigurációban az iterációk között kiürítem a gyorsítótárat minden alkalommal, másik konfigurációban nem. A várt eredmény az, hogy utóbbi esetben az első pár iteráció közel megegyezik az első konfiguráció eredményeivel, viszont utána a cache jelenléte miatt gyorsabb lesz.

Raspberry Pi 3B+ eszközön, kétezer iteráció alapján a ciklusok egy 5 százalékos sávban helyezkednek el. Többnyire a gyorsítótár használatával kevesebb idő kell a program lefutásához, viszont akadnak bőven olyan esetek is, ahol fordítva van.



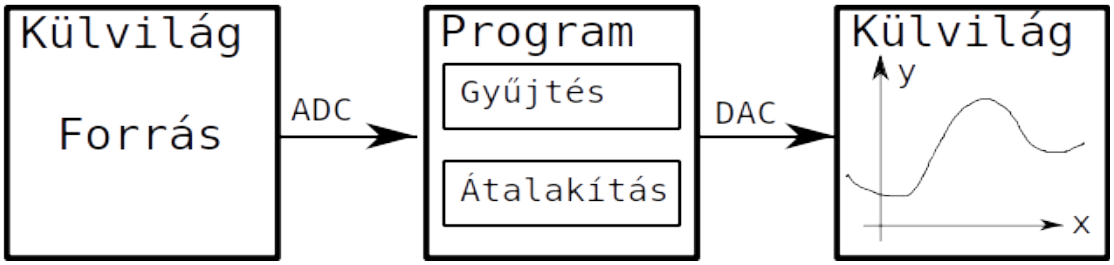
34. Ábra: Gyorsítótár hatásának vizsgálata Pi 3B+ eszközön

A „warmup-iterations” jelenséget viszont ezzel a méréssel nem sikerült előállítanom. Második próbálkozásként egy FIR Neon implementációt vizsgálok meg, hátha a vektorokkal jobban kihasználja a gyorsítótárat a program. A vizsgálni kívánt jelenséget itt sem sikerült előidézni, a mérés eredménye az előzővel megegyezik. Gyorsítótár használatával kevesebb idő kell a program futásához, de nem sikerült kimutatni az első néhány iteráció során jelentős eltérést. Ennek oka az lehet, hogy kis méretű a program és a gyorsítótárnak sikerült már az elején alkalmazkodnia hozzá.

8 Valósidejű használat

Bemenetként az USB portra csatlakozó hangkártyából kapott jeleket dolgozom fel és a feldolgozás eredményét továbbítom a választott hangkártyának. Szoftveresen csak C nyelven implementálom a keretet, ami a fentebb megírt algoritmusokat használni képes Raspberry Pi 3B+ és 2 eszközzel.

A modell adatok gyűjtéséből és feldolgozásából áll. Az architektúrát kihasználva a modell különböző szálai saját magokon futnak, amelyek a többi „user-space„ folyamatoktól izolálva vannak. Előnye, hogy egy mag csak egy felhasználói feladatot végez. Az izoláltságnak hátránya is van, mégpedig a feldolgozó egység magja nem fér hozzá az adatgyűjtő magjához, így a gyorsítótár használata nem annyira hatékony, mintha egy magon futnának és az alacsony szintű gyorsítótáron osztoznának. További hátrány még, hogy „real time kernel” esetén a CPU affinitást nem lehet állítani, ezért a program futtatható izolált magokkal és nem izolált magokkal. Az előző fejezet mérése alapján nem is feltétlenül célszerű az általam fordított kernellel jelet feldolgozni. Érdemesebb egy olyan kernelt használni, ahol lehet magokat izolálni és az ütemezőre tudjuk bízni a szálakat.



35. Ábra: A rendszer blokkvázlata

A hangkártya programozásához az ALSA[4] driver csomagot használom, ezért ennek a fejlesztői könyvtárait külön kell telepíteni az eszközökre (*libasound2-dev*

[4]Advanced Linux Sound Architecture, egy driver csomag amivel alacsony szinten lehet a hangkártyát programozni pulzus kód modulációval.

csomag). Az ALSA felkonfigurálását elvégző függvényt az ALSA dokumentációja és a Linux Journal leírása alapján készítettem el. [28]

Egy tetszőleges hangkártya használatához kettő struktúra inicializálása szükséges. Kell egy struktúra, ami magát a hangkártyát reprezentálja, valamint egy struktúra a hangkártya paramétereinek. A hangkártya paramétereinél kell megadni, hogy az eszköz lejátszó vagy felvevő, csatorna beállítását (mono vagy sztereó) és a csatorna módját amennyiben sztereó. A csatorna módja jelenleg átlapolt, vagyis a két csatorna mintái egymást követik felváltva. Egyéb csatorna beállításokat is meg lehet itt adni, mint például periódusidő tartamát.

A fordítás az alábbi paranccsal lehetséges. A kódban zölddel vannak jelölve az architektúra függő és vektorizációért felelős részek, kékkel a könyvtárspecifikus részek.

```
aarch64-linux-gnu-gcc -O2 -ftree-vectorize process_signals.c -ffast-
math -march=armv8.2-a+fp16 -lpthread -D _GNU_SOURCE -lasound -
lwiringPi

arm-linux-gnueabihf-gcc -O2 -ftree-vectorize process_signals.c -
ffast-math -lpthread -D _GNU_SOURCE -lasound -lwiringPi
```

8.1 Adatok gyűjtése

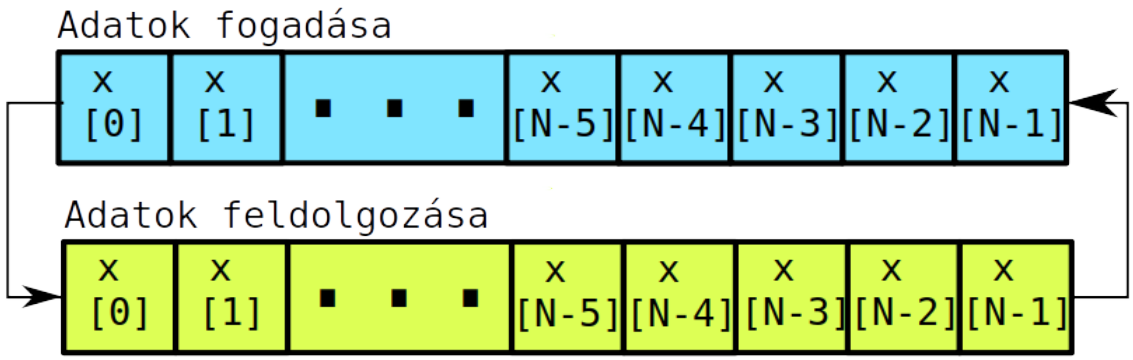
Az előjel nélküli 16 bites adatokat egy külső forrásból gyűjtöm be egy USB átalakító segítségével. Az adatokat egy szál fogadja és tárolja el. Közös memóriaterületet használok, szálak lévén közös folyamathoz tartoznak, így el tudják érni ugyanazokat a memóriarészeket. Mivel az USB-s hangkártya csak 16 bites előjeles egészeket képes fogadni és küldeni, ezért az adatgyűjtés és feldolgozás is ilyen típusokkal dolgozik elsősorban. Implementálva van lebegőpontos típussal való feldolgozás is, ami a bejövő egészeket alakítja át lebegőpontosra, majd feldolgozás után visszaalakítja, hogy a DAC tudja kezelni az adatokat.

Meg kell bizonyosodni arról, hogy nem írunk felül olyan tárolt bemeneti adatot, amit még a feldolgozó szál használni akar, ezért az adatoknak megfelelő struktúrát kell választani és védeni a felülírástól. Kettős tárolót használok ezért, aminek nagy előnye, hogy ameddig az egyikbe írunk, addig a másikból olvasunk. A folyamat végén csak egyszerűen fel kell cserélni őket. Egyik hátránya, hogy ameddig fogadunk adatokat egy blokkba, addig azt a blokkot nem írhatjuk ki, hiába lehet már néhány adata feldolgozva. Másik hátránya a többszörös memóriaigény lenne, de az eszköz esetén ez nem jelent

gondot. A feldolgozásnál szükség lenne egy harmadikra, hogy az előző adatok rendelkezésre álljanak, viszont ehhez egy ugyanolyan méretű tömb pazarló lenne. Az előző bemeneti (és IIR szűrő esetén kimeneti) adatokat viszont mindenképpen fel kell használnom, ezért azoknak egy olyan tömböt foglalok, amelynek mérete megegyezik a szűrőfokszámmal. A régebbi kimeneti értékeket is el kell tárolni IIR szűrő esetén, de az alacsony fokszámú IIR szűrő miatt mindig a meglévő kimeneti tárolót használom mindig a szűrő fokszámával eltolva, aminek a mérete ebből adódóan egy szűrőfokszámnyival nagyobb. Az elv az, hogy a kimenet első szűrőfokszámnyi adata az előző adatsor utolsó szűrőfokszámnyi adata hátulról feltöltve, amennyiben érvényes. Amennyiben nincsen ennyi érvényes adat, akkor nullákkal egészítem ki a tömb eleje felé haladva. Az a gondolatmenet érvényes a bemeneti adatok kezelésére is. Tervezési szempont még a beolvasás és a feldolgozás sebességét egymáshoz igazítani: ne alakulhasson ki olyan akadály, hogy az egyik szál „sokat” vár a másik szálra.

A szál egy inicializáló részből és egy beolvasó ciklusból áll. Az inicializálás során opcionálisan lefoglaljuk a szál számára a kijelölt magot és amennyiben rendelkezésünkre áll bemeneti hardver, a hangkártyát is felinicializáljuk.

A program tesztelését hangkártya nélkül is elvégeztem pszeudo-véletlen számokkal, amiknek forrása */dev/urandom*, tesztjellel és felvett hanganyaggal is. Továbbá lehetséges a program tesztelése beépített négyszögjellel és képes fájlból is olvasni a bejövő adatokat. Időzítési viszonyokat ezzel nem lehet vizsgálni, mert gyorsabb és egyszerűbb, mint egy AD konvertálás, viszont a programvezérlést és a szinkronizálást egyszerűbben és kényelmesebben lehet tesztelni vele. Az inicializálás végén, mielőtt elindítanánk a beolvasást bevárjuk a másik szálat.



36. Ábra: Kettős tároló használata

A szál egy ciklusba kerül, ahol folyamatosan beolvassa az adatokat az egyik tárolóba. Miután befejeződött a blokk beolvasása, az olvasásra használt mutatót ráállítom az éppen beolvasott tömbre. Az írásra szánt mutatót átirányítom a következő szabad tárolóra. Jelen esetben összesen kettő tároló van, így csak egy lesz szabadon írható.

Ha valamilyen módon kilépnénk a ciklusból, akkor az erőforrásokat felszabadítja a szál, majd várakozik a feldolgozó szál kilépésére. Pszeudokóddal leírva a szál működése hangkártyás működés esetén:

```
CPU_affinitás_beállítása()
Hangkártya_inicializálása_bemenetként()

Iteráció_szám = 0

Várakozás_a_feldolgozó_szál_inicializálására()

Ciklus amíg van adat:
     Státusz = Beolvasás_hangkártyával()
     Hibakezelés_státusz_alapján()

     Régi_tömb_mutató = Következő_tömb_címe[++Iteráció_szám%2]
     Új_tömb_mutató = Következő_tömb_címe[++Iteráció_szám%2]

Erőforrások_felszabadítása()
```

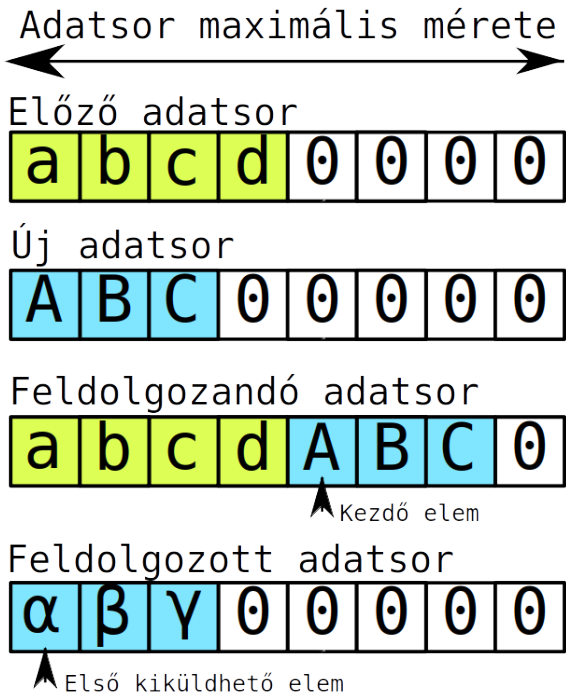
A ciklusban az elágazásokat kikerülve választom ki az új tömböt, amibe a szálnak szabad adatot beolvasnia. Ehhez egy tömböt veszek fel, amiben a kettős tároló egy-egy mutatóját tárolom el. Az új, szabad tömböt az iterációszám és modulo kettes osztás segítségével már könnyedén meg lehet kapni elágazás nélkül. A bejövő adatok tesztelésére kialakítottam egy makrót, amivel a feldolgozandó adat helyett a nyers bejövő adatot adom ki a DAC-n.

8.2 Adatok feldolgozása

A bejövő adatokat fel kell dolgozni valamilyen implementált algoritmus szerint, majd továbbítani a DAC felé. Eddigiekhez hasonlóan fordítási időben az algoritmus lecserélhető egy a rendszerrel kompatibilisre.

A feldolgozás szempontjából figyelembe kell venni, hogy az elemek függhetnek az előző értékektől. Ez akkor kritikus, ha az új blokkunk első néhány eleme igényel olyan adatokat, amelyek a blokknak már nem elemei. Az előző adatsort el kell tárolni, így itt szükségünk van egy plusz tárolóra. Kényelmi okokból folytonosan érdemes

tárolni az adatokat, mégpedig feldolgozási sorrendben. A gyakorlatban ez azt jelenti, hogy a feldolgozásnál egy olyan tömböt használok, ami kettő másik tömbből áll össze. Mivel egy beolvasás során nem feltétlenül töltjük meg a tárolónkat adatokkal, hanem akár jelentősen kevesebb adattal is, ezért az előző és az új adatok egymáshoz való illesztése problémás. A hangkártya továbbra is konstans méretű tárolóval dolgozik. Az általam használt megoldás arra támaszkodik, hogy pontosan nulla adatot nagyon ritkán olvasunk be hardverből, mert mindig van valamennyi zaj. Lebegőpontos számok esetén számmal való egyenlőségvizsgálat a pontosságtól függ és rossz gyakorlat, ezért szükséges egy jobb megoldás keresése. A régebbi adatok közül csak az utolsó szűrőfokszámnyi adat kerül hasznosításra. Ettől illesztem hozzá az új adatokat. A feldolgozást addig érdemes végezni, ameddig van mit. A felesleges nullákat nem célszerű feldolgozni a hasznos adathalmazzal, viszont a kimenetre mindig ugyanannyi keretet kell kiadnunk azért, hogy a frekvencia ne torzuljon. Tehát a feldolgozás során kapott eredményeket ki kell egészítenünk megfelelő számú nullákkal. A következő ábra egy olyan esetet reprezentál, ahol az adatsor maximális mérete nyolc egység és a szűrő fokszáma négy.



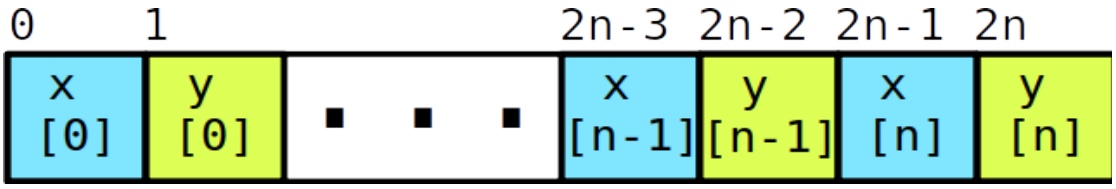
37. Ábra: A feldolgozás menete a meglévő adatok használatával

A feldolgozott adatokat ALSA könyvtár segítségével a program tovább küldi a DAC-nek, amit már lehet PiScope, logikai analizátor vagy akár valós oszcilloszkóp segítségével vizsgálni.

Az eszközről „low endian” 16 bites előjeles számokat fogok küldeni. Ez nem azt jelenti, hogy nem dolgozhatok lebegőpontos számokkal, csak a beolvasást és a kiküldést korlátozza. Lebegőpontos számábrázolás esetén is 16 bites előjeles egészeket olvasunk és küldünk ki, viszont az algoritmus 32 bites lebegőpontos vektorokkal dolgozik. Ezzel a futási idő lehetséges növelésével a pontosságon javítunk. Előjeles számok esetén az algoritmus nyolc darab 16 bites előjeles egésszel dolgozik, ezért a koefficienseket is át kell alakítani előjeles egésszé. IIR szűrők esetén itt felvetődik az a kérdés, hogy az osztás hogyan valósítható meg úgy, hogy ne torzuljon az eredmény. A koefficienseket úgy kell megadni, hogy az osztónk értéke egy legyen. Mivel fordítási időben ismernünk kell a koefficienseket, ez átalakításokkal mindig megtehető. Ez akkor lehetséges, hogyha nincsen egynél nagyobb koefficiensünk. Egyéb esetben a legnagyobb értékeket felvevő koefficiensre kell normalizálni az összes többit és aztán átalakítani, hogy ne legyen túlcsordulás az átalakítás során. Az értékek így erősen torzulnak és ezzel hibát viszünk be a mérésünkbe, valamint az osztással sem lesz mindig maradék nélküli, ami további hibát vet fel. Octave programmal a következőképpen lehet kiszámolni a keletkező kerekítési hibát:

```
(koefficiensek – round(koefficiensek*2^15)/2^15)./koefficiensek)
```

Az eszközt sztereó kimenetűnek konfiguráltam fel, mert a DAC is sztereó. Sztereó csatornákat kihasználva egyik csatornán az eredeti kapott jelnek minden második elemét továbbítom, másik csatornán a már feldolgozott jelet. „Interleave” módot használva a kimenetre a minták párosával kerülnek ki: először a minta amit az első csatornára akarunk küldeni, aztán a minta amit a második csatornára küldünk. Megvalósítás szempontjából ez annyit jelent, hogy az első csatornára csak páros, a második csatornára csak páratlan indexeket használok az adatok eléréséhez a keretekben, valamint fele annyi adatot olvasok be bemenetről, mint amennyit kiküldök. Makrótól függően ez a megoldás érvényes, vagy sztereó módban a feldolgozott adatokat vagy sztereó módban a még fel nem dolgozott adatokat küldjük ki. A tároló periódusokra van osztva, amikben keretek vannak. Amennyiben megtelik a tároló, egy megszakítás jön létre ami utasítja a kernelt az új adatok beolvasására DMA-val, a processzor megkerülésével.



38. Ábra: A bemenet ("kék" csatorna) és a kimenet ("sárga" csatorna) küldése a külvilágnak külön csatornákon egy kereten belül.

A [28]. cikkben található egyszerű mintapéldával ellenőriztem, hogy a kód ténylegesen kiküldi-e az adatokat a hangkártyának vagy sem. A mintapélda tesztelése során használtam GPIO portokat is nyomkövetés céljából. Például arra használtam egyet, hogy az adatküldést tényleges időpontját láthassam, egy másikat a hibaeseményeknek tartottam fent. A szál működése hasonló a másik szál működéséhez, pszeudokóddal leírva a következő:

```
CPU_affinitás_beállítása()
Hangkártya_inicializálása_kimenetként()

Várakozás_az_olvasó_szál_inicializálására()

Ciklus amíg van adat:
     Kapott_adatok_feldolgozása()
     Adatok_szétosztása_csatornákra()

     Státusz = Küldés_hangkártyával()
     Hibakezelés_státusz_alapján()

Erőforrások_felszabadítása()
```

Tesztelése a következőképpen zajlik: hoszt gépen valamilyen hangot adok ki, amelyet továbbítok a nyákon keresztül az USB-s hangkártyának. A Pi oldalán ezt ALSA segítségével felveszem és fájlként beadom a programnak, vagy közvetlenül kezdem el feldolgozni. Tesztadatsorként 1500 Hz-es és 5000 Hz-es szinusz jelet használtam, amit a „speaker-test” segítségével hoztam létre.

```
speaker-test -t sine -f 1500
```

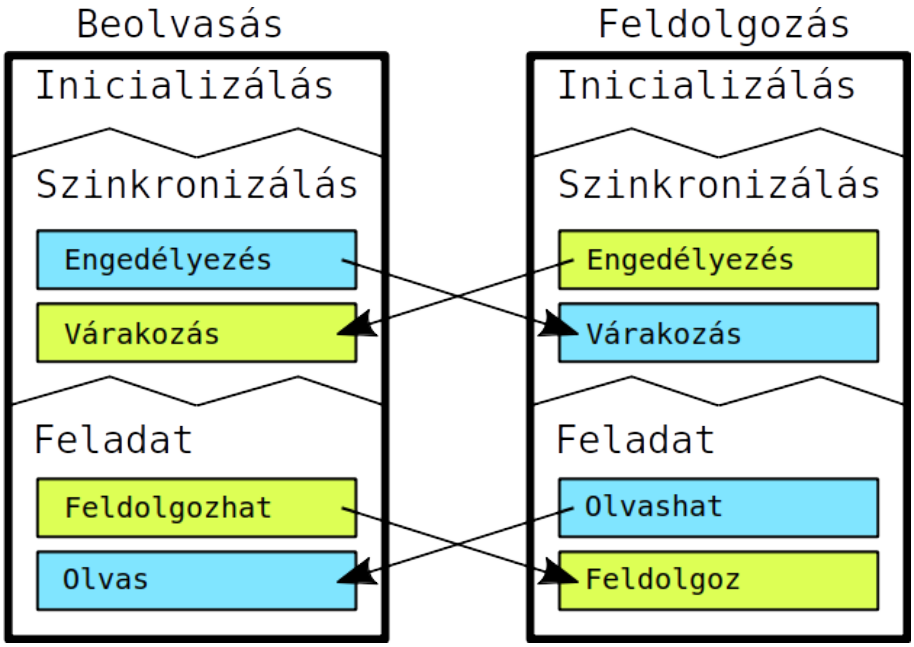
8.3 Adatok szinkronizálása

Megfelelő szinkronizációval elkerülhetjük azon hibás eseteket, amikor az adat nem megfelelően kerül ki a kimenetre. Szeretnénk megelőzni azt, hogy még ki nem

küldött adatokat írjunk felül az új adatainkkal vagy hogy olyan adatot küldjünk ki újra, amit már egyszer kiküldtünk. Előbbi esetre nyújtanak megoldást a kettős tárolók, utóbbira a megfelelő mintavételi frekvencia és időzítések használata.

A kivitel sebességét a beolvasás sebességéhez igazítom a hangkártya felkonfigurálásánál. Jelen esetben 16 bites bemeneti adatokat legfeljebb CD minőségben, vagyis 44100 Hz frekvenciával tudok szolgáltatni, ezért a kimeneti frekvencia is ennyi lesz a feldolgozás során. 8000 Hz frekvenciával tesztelve a kapott jel minősége jelentősen rosszabb volt, mintha magasabb frekvenciával mintavételeztem volna.

Előfordulhat, hogy valamelyik szálon a feladat gyorsabban befejeződik, mint a másikon. Ekkor a szál alapesetben újrakezdené a feladatát. Pszeudo-véletlen számokkal való tesztelés során vetődött fel az a probléma, hogy az értékek elhelyezése a tömbben lényegesebben gyorsabb volt, mint a feldolgozó szál feladata. Ekkor akár három „beolvasás” is végbe ment, miközben a feldolgozó még ki sem adta az első eredményét. Ennek elkerülése érdekében a POSIX könyvtár elemeit használom a szálak szinkronizálására. Elsőként a hangkártyák inicializálása után várják be egymást a szálak, hogy „nagyjából egyszerre” kezdhessék a feladatukat. A feladatuk végzése közben is szükségük van a szinkronizációra: elkerülendő az, hogy a kettős tárolóból felülírjunk olyan adatsort, amit még nem dolgoztunk fel. Addig nem írunk a még fel nem dolgozott tömbre, ameddig a feldolgozásnál ezt nem engedélyezzük. A mutexeket és a feltételes változókat még a szálak létrehozása előtt inicializálom és a szálak megszűnése után semmisítem meg őket. Mivel rossz gyakorlatnak minősül a mutexek lezárása és másik szálban való feloldása, ezért feltételes változókkal és a hozzájuk tartozó „volatile” változókkal engedélyezem és tiltom a szálakat, amik használata szálon belül mutexek segítségével történik.



39. Ábra: Szálak szinkronizálása feltételes változók segítségével

Szintén a szinkronizáláshoz kapcsolódik a szálak nyomkövetésének egy esete. A sztenderd kimenet mint erőforrást használhatja egyszerre több szál is, ezért előfordulhat olyan eset, hogy a kimeneten a szálak által kiírt karakterek összefésülődnek. Ennek kizárására biztosítani kell, hogy az erőforrást egyszerre csak egy szál tudja használni.

8.4 Konklúzió

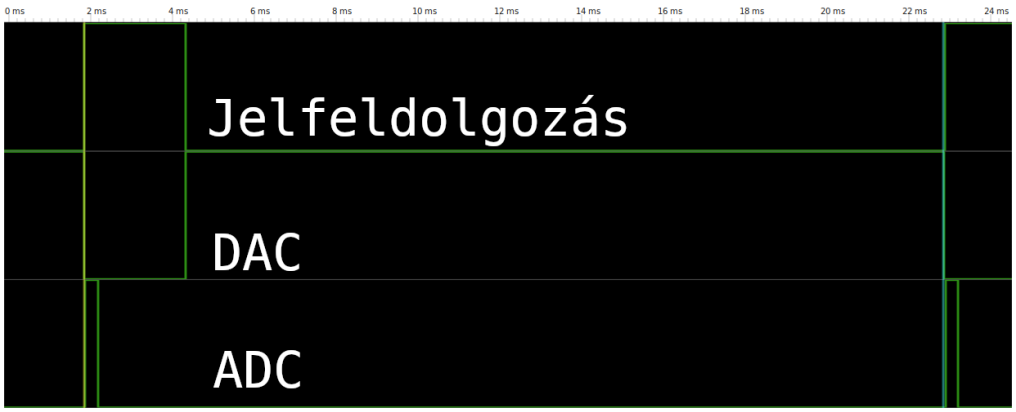
8.4.1 Időzítési viszonyok

A szálakat a kernel ütemezi, ami konfigurációtól függően eltérhet, akár külön magon is futhatnak és lehetnek egyedüli felhasználói szálak. Inicializálásuk után bevárják egymást, hogy ne kezdjen el egyik sem „sokkal” a másik előtt dolgozni, majd egymással összehangolva olvassák és dolgozzák fel az adatokat. Erre azért van szükség, mert látszólag az ADC általi beolvasás a töredéke a DAC általi kiküldésnek.

A beolvasás és adatok kiküldése egy USB-s hangkártyával történik, mindkettő feladatot ALSA könyvtár segítségével valósítom meg. A beolvasás és feldolgozás során kettő tárolót használok és blokkonként dolgozom fel az adatokat, viszont feldolgozásnál mindig az előző blokkot dolgozom fel, ezért legalább egy blokknyi késleltetése van.

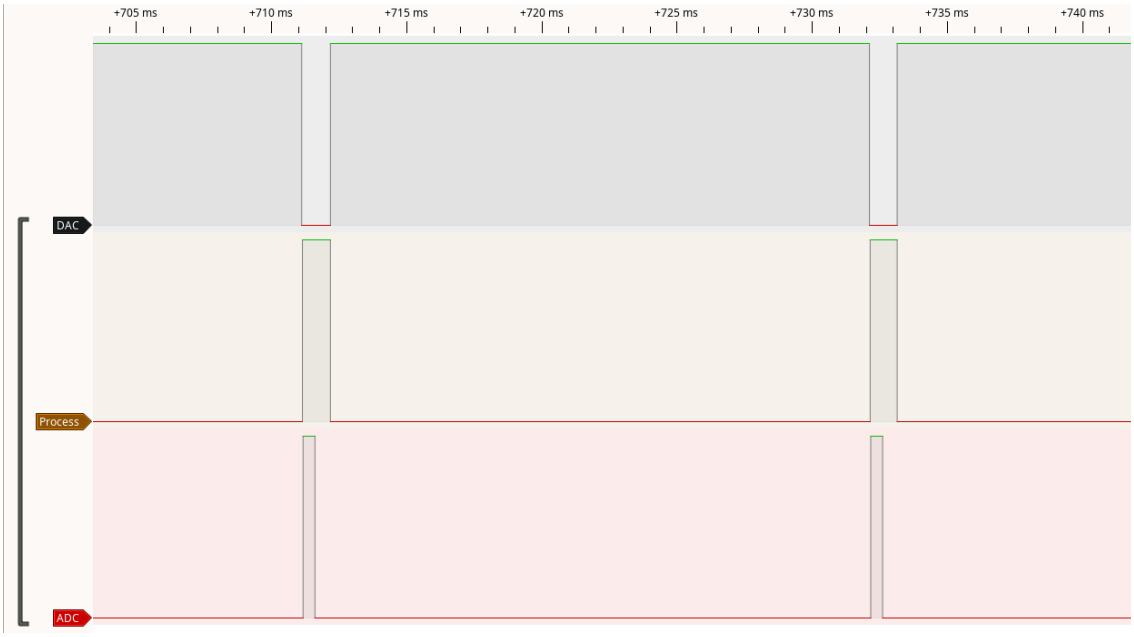
A tesztelést Pi 2 és 3B+ modellen végeztem el PiScope és oszcilloszkóp segítségével. Néhány GPIO lábat nyomkövetésre állítottam be és a mérendő kódrész előtt és után értéküket változtattam. A méréshez szükséges csomag 64 bites kompatibilitásának hiányában csak a hagyományos implementációkat mértem le 32 bites operációs rendszeren, Ubuntu 18.04, 5.3-as kernelen.

A mérések alapján 52 fokú FIR és 5 IIR szűrő esetében is kicsivel gyorsabbnak bizonyult a lebegőpontos implementáció. Általánosságban 20-21 milliszekundum kell egy teljes beolvasás-feldolgozás-kiküldés feladatsornak Raspberry Pi 2 eszközön. Az ADC feldolgozás valószínűleg nem blokkoló függvény a DAC feldolgozáshoz képest. Ez utóbbi nagyjából 18,5 milliszekundum, egy nagyságrenddel kisebb. Raspberry Pi 3B+ eszközön a teljes beolvasás-feldolgozás-kiküldés ciklus 6 milliszekundum körül van 1.4 GHz CPU frekvencia mellett.



40. Ábra: Adatok fogadása, feldolgozása és kiküldése Raspberry Pi 2 eszközön FIR szűrő esetén PiScope programmal

Logikai analizátor segítségével is mérhetjük a futási és feldolgozási időket. Az általam rendelkezésre állt eszközzel mintavételezési frekvenciától függően pontos eredményeket kaptam, annak ellenére, hogy ugyanúgy a GPIO lábakat mérem közvetlenül vezetékekkel. Például egy IIR szűrő a Raspberry Pi 2-es modellen 900 MHz működési frekvenciával és 5 MHz mintavételi frekvenciával mérve:

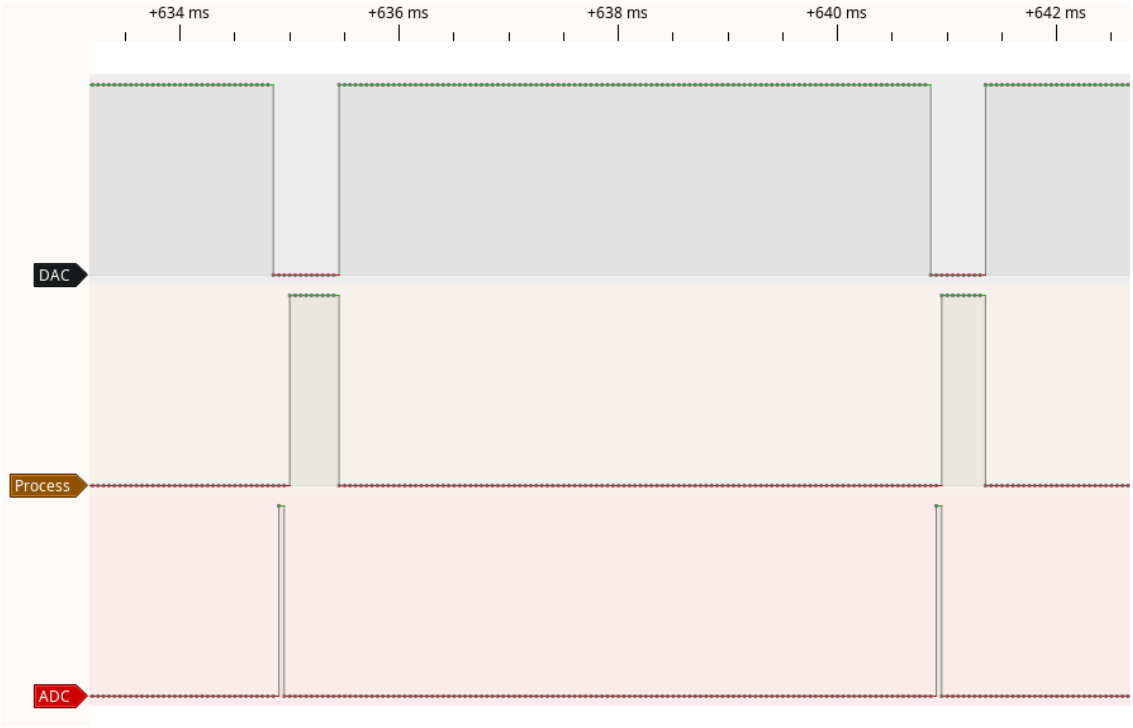


41. Ábra: Futási idő mérése valós szűrés közben Pi 2 eszközön Pulseview program segítségével

A mért értékek alapján nehéz levonni a tanulságokat, ugyanis a mintavételezési frekvencia szabja meg a pontosságát. 20 kHz mintavételi frekvenciával mintavételezve a GPIO lábakat sokkal torzultabb eredményeket kapunk, mint 24 MHz frekvenciával. A feldolgozó implementációjából adódóan nem lehet egyszerűen a futási értékeket mérni, hiszen amennyiben van hardver, amiből tudunk beolvasni adatot, sosem rögzített méretű adatokat dolgozunk fel. Ez a beolvasás hibájából adódik, hogy nem képes mindig teljes kereteket beolvasni. Az eltérő keretszám az ADC futási idejéből nem, viszont a feldolgozásból kimutatható. A feldolgozás monitorozása során előfordult, hogy hagyományos definíció alapú FIR szűrő esetén gyorsabban történt a feldolgozás, mint Neon alapúval. Ez a korábbi méréseim alapján azonos adatszámmal nem lenne helytálló, 52 szűrőegyüttható́val a vektorizáció is megérné. Amennyiben a feldolgozás idejét szeretnénk mérni érdemes a rendelkezésre álló négyszögjelet vagy pszeudo-véletlen számokat használni azért, hogy mindig ugyanannyi adatot dolgozzunk fel.

Általánosságban igaz mindkettő szűrőre, hogy megéri lebegőpontos számokat használni előjeles egészek helyett, a feldolgozás gyorsabban végbemegy velük. Ennek látványos hatása IIR szűrők esetén van, ahol Neon esetén előjeles számokkal dolgozva 442 µs mikroszekundum, lebegőpontos számokkal dolgozva 242 µs volt a feldolgozás. Amit még érdemes megjegyezni, hogy valóban ötödfokú IIR szűrő esetén még gyorsabb

volt a definíció alapú, mint a vektorizált, még ha csak 3 százalékkal is. FIR szűrő esetén a magas fokszám miatt jobban megéri a vektorizált implementációkat használni lebegőpontos számokkal. Ami érdekes, hogy Neon esetén még úgy is gyorsabb a lebegőpontos számokkal való feldolgozás, hogy négy darab 32 bites lebegőpontos szám alkotja a vektort, másik esetben pedig nyolc darab 16 bites előjeles egész.



42. Ábra: Teljes feldolgozási ciklus Pi 3B+ eszköz esetén definíció alapú FIR szűrővel

Előfordulhatnak olyan esetek, amikor az ADC hibásan olvas be. Ilyenkor egy hibaüzenettel jelezzük a megfelelő kimeneten, valamint várnunk kell a következő beolvasásra. Valós idejű feldolgozás szempontjából ez kellemetlen, tudjuk, hogy jönni fog adat előbb-utóbb, de nem tudjuk megmondani pontosan, hogy legfeljebb hány ilyen hibás beolvasási ciklus lehet. Fájlból (vagy „streamből”) való beolvasás során ez a probléma nem áll fenn, mert rendelkezésre állnak a minták.

A DAC is képes hibás működésre, amit beépített függvények kezelnek. Ezeknek a függvényeknek is van „overheadje”, a hibák jelzését a feldolgozás során opcionálisan külön erre beállított GPIO lábakkal jelzem.

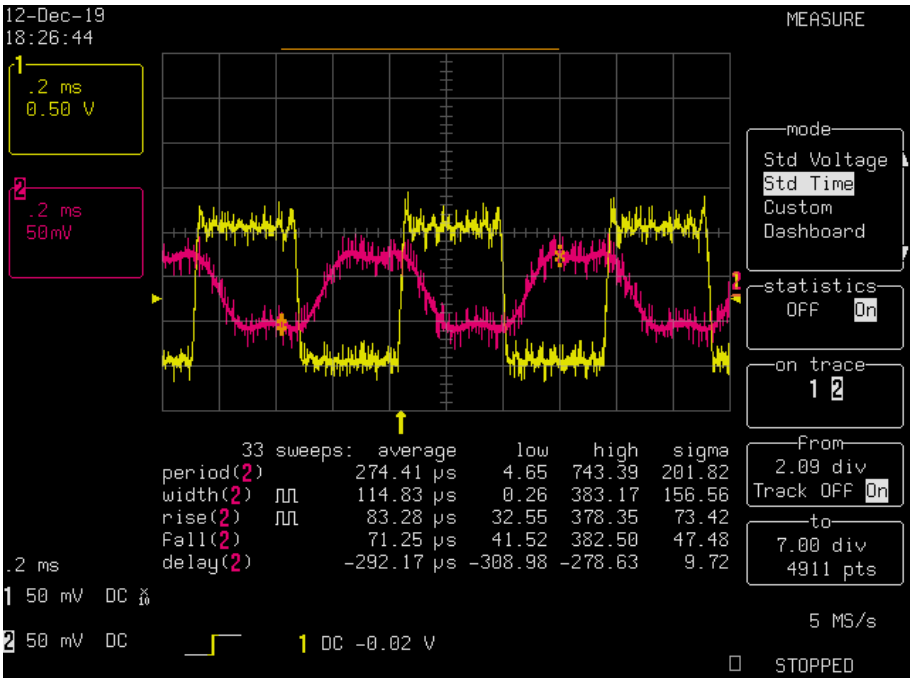
Az eszköz nyomkövetése is az ilyen lábakra támaszkodik, mivel szükség van arra, hogy a futásbeli időzítési viszonyoknak megfelelően lehessen a programot

elemezni. Más adatokkal dolgoznánk akkor, ha leállítanánk a mérést, mintha folyamatosan mérnénk.

8.4.2 Kimenet vizsgálata

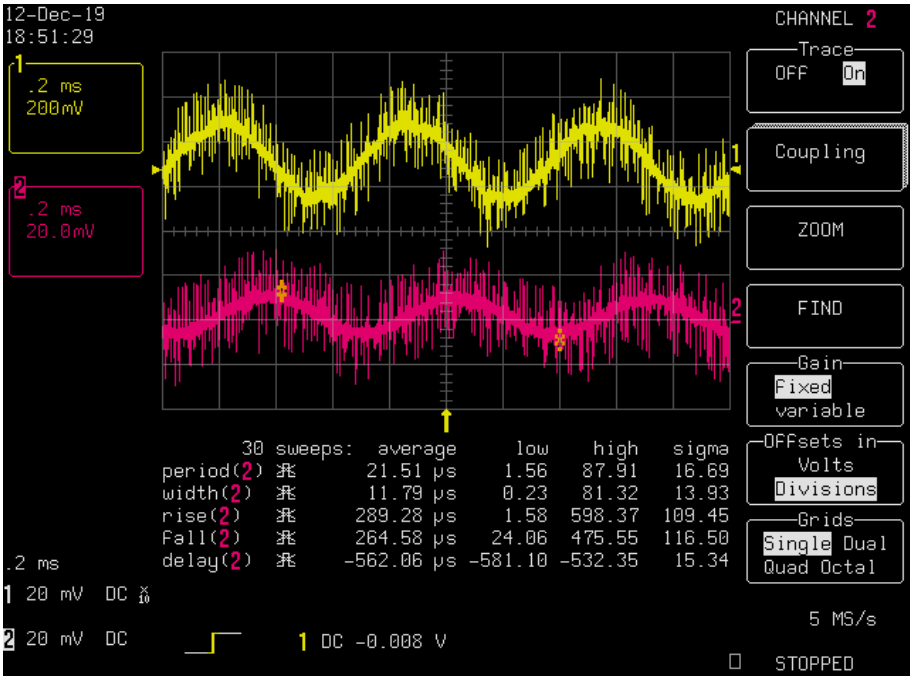
Annak ellenére, hogy a feladatom az egyes jelfeldolgozási algoritmusimplementációk hatékonyságának összehasonlítása volt, azt is leteszteltem, hogy az implementált szűrési műveletek valóban működőképesek-e. Különböző vizsgálójeleket alkalmazva figyeltem a be- és kimeneti jeleket.

Az első mérés során, a 43. ábrán, egy általam a programba beépített négyszögjel látható az egyes csatornán (sárgával), a kettes csatornán a Neon alapú FIR szűrővel feldolgozott jel (rózsaszínnel). A kimenet rendkívül zajos és késleltetése a bemenethez képest mérhetően nagy.



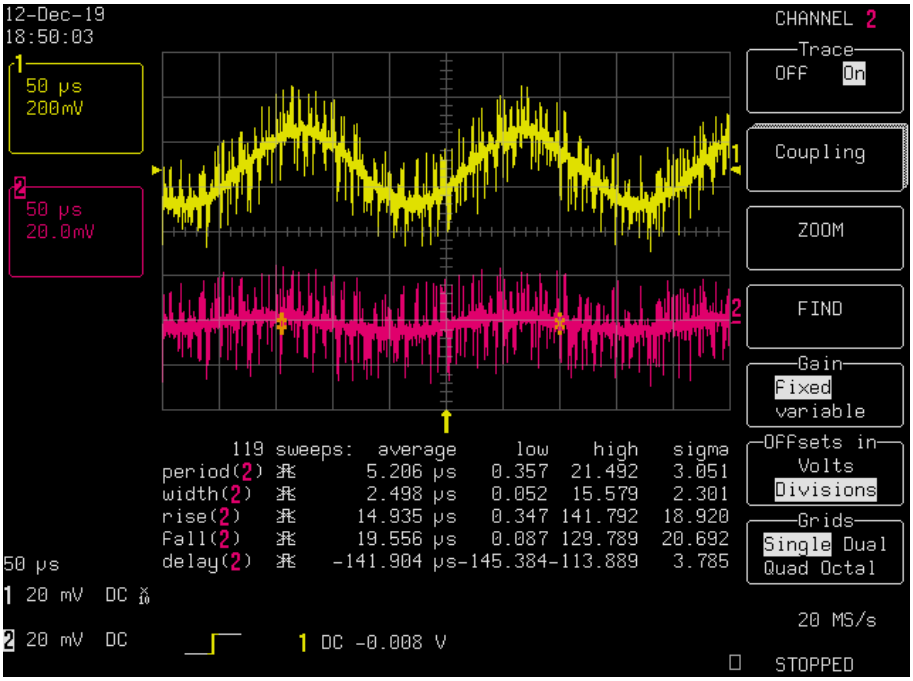
43. Ábra: Beépített négyszögjel szűrése ötvenketted Neon alapú FIR szűrővel

Második mérésnek (44. ábra) egy külvilágtól érkező jelet dolgoztam fel. Egy 1500 Hz-es szinusz jel látható a képen, amit a szűrőnek még át kell engednie csillapítás nélkül. Az előző ábrához hasonlóan az egyes csatornán az eredeti jel, kettes csatornán a hagyományos IIR szűrővel szűrt jel látható.



44. Ábra: Külvilágtól érkező 1,5 kHz frekvenciájú szinusz szűrése ötödfokú hagyományos IIR szűrővel

Harmadik mérésként (45. ábra) egy 5000 Hz frekvenciájú szinusz jelet használtam, ahol már csak a csillapított jelet kellene látnunk. A szűrő megegyezik a második mérésben használt hagyományos IIR szűrővel. Itt a zavarok nagysága már meghaladhatja a kimenet „peak-to-peak” amplitúdóját is.



45. Ábra: 5000 Hz frekvenciájú szinusz jel szűrése hagyományos ötödfokú IIR szűrővel

9 Értékelés és kitekintés

A jelenlegi hangkártyás megoldás a kiegészítő áramkörrel működőképes, viszont a minősége túlságosan ingadozó. Ahhoz, hogy a gyakorlatban tényleg használható legyen a jelenlegi programom, egy jobb, zajmentesebb megoldást kell alkalmazni, akár jobb hardvereset, akár más szoftvereset.

A jeleken ülő jelentős zajt valószínűleg egyrészt a nem túl jó minőségű hangkártya okozza, másrészt a mérés során több eszköz is külön tápfeszültségről működött, így ezek összekötése sok zavart visz a rendszerbe főleg a kapcsoló üzemű tápegységek miatt.

Késleltetése a kettős tároló miatt és a jelenlegi konfigurálása miatt kellemetlenül sok. Egyszerű jelátvitel esetén néhány másodperc volt, amíg a rendes minták bekerültek a tárolóba és a megfelelő jel alakult ki a kimeneten, annak ellenére, hogy 20 ms körüli egy beolvasás-feldolgozás-kiküldés ciklus Pi 2 eszközön, 6 ms körüli 3B+ eszközön. Ez megfelelő konfigurálással, mintaszámmal és talán jobb adatstruktúrával lehetségesen csökkenthető lenne. Komolyabb, idő kritikusabb alkalmazásokhoz ez az összeállítás jelenleg ilyen formában nem ajánlott, alkalmatlan.

Egy harmadik szál bevezetésével a feldolgozási időt megspórolhatjuk, külön magot is foglalhatunk le csak a feldolgozásnak, viszont akkor mindig eggyel régebbi adatsorhoz tartozó adatokat kellene kiküldenünk. Összességében a beolvasás-feldolgozás-kiküldés ciklus rövidebb lenne annyival, mint amennyi a feldolgozáshoz szükséges lenne, viszont később küldenénk ki a külvilágnak a feldolgozott jelet. Hátránya lenne még a szinkronizáció megoldása.

Másik hibája a jelenlegi feldolgozó rendszeremnek, hogy Raspberry Pi 3B+ esetén kétszer annyi keretet akar beolvasni a mikrofonból, mint amennyit kellene. Ez a mellékhatás az operációs rendszernél a helytelen konfigurálás miatt léphet fel, Pi 2 esetén az elvártnak megfelelően működik a C-s konfigurálással. A mellékhatást szoftveresen lehet javítani a keretek egyszeri felezésével még a beolvasó ciklus előtt, ha makróval engedélyezzük. Meg lehetne valósítani azt is, hogy fájlként megnyitjuk a modell fájlját és kiolvassuk milyen modellről van szó a makró elkerülése érdekében,

viszont a probléma okának tényleges megértése nélkül célszerűtlen lenne minden egyes Pi 3B+ eszközhöz ezt a funkciót hozzárendelni.

Választhatunk egy olyan hangkártyát, ami egyben ADC és DAC is. Az általam tesztelt megoldás során a teljes beolvasás-feldolgozás-kiküldés ciklusnak a kritikus részét a kiküldés okozta, amin nem tudtam szoftveresen javítani. Ez kiküszöbölhető lenne egy jobb DAC-vel. Illeszthetünk külső modulokat is, amik nem feltétlenül a Pi családhoz lettek készítve. Még egy ilyen megoldás az Ethernet kapcsolaton keresztüli adatküldés lenne, ahol már digitális adatot fogadunk és küldünk az eszközöknek.

A tesztadatsor és a véletlen számok használata zökkenőmentesen működik mindkettő szűrőalgoritmussal, viszont tesztadatsor hátránya, hogy csak fordítási időben változtatható meg. A fájlolvasás hátránya maga a fájl olvasása, vagyis a fájl mérete adattal való bővítése során folyamatosan nő, így ez lemezterületbe kerül, vagy ismernünk kell az összes bemeneti adatot. Stream esetén ez a hiba kiküszöbölhető. A „wav” hangfájlok feldolgozása zökkenőmentesebb, de ennek hátránya, hogy előre rendelkezésünkre kell álljanak a bemeneti adatok megfelelő formátumban.

A feldolgozás során méréseim alapján érdemes mindig lebegőpontos számokat és Neon implementációkat használni FIR szűrő és magasabb fokszámú IIR szűrő esetén C nyelven implementálva. Ötödfokú IIR szűrő esetén a definíció alapú még gyorsabbnak bizonyult néhány százalékkal. Gyorsítótár használata általánosságban ajánlott, javít a teljesítményen, még ha csak a koefficienseknél tudjuk alkalmazni. Léteznek külső könyvtárak, amik kifejezetten jelfeldolgozásra lettek optimalizálva, érdemes lehet azokat is összehasonlítani és alkalmazni. Azok használatával valószínűleg a feldolgozási idő is csökkenthető lenne.

Irodalomjegyzék

[1] Diamond Systems: COM-Based SBCs: The Superior Architecture for Small Form Factor Embedded Systems, http://whitepaper.opsy.st/WhitePaper.diamondsys-combased-sbcs-wpfinal-.pdf (2019. szeptember)

[2] The Raspberry Pi Foundation, About Us, https://www.raspberrypi.org/about/ (2019. szeptember)

[3] Wikipedia, Banana Pi, https://en.wikipedia.org/wiki/Banana_Pi, (2019. szeptember)

[4] Wikipedia, BeagleBoard, https://en.wikipedia.org/wiki/BeagleBoard, (2019. szeptember)

[5] OnionDOCS, Introduction to the Omega2, https://docs.onion.io/omega2-project-book-vol1/omega2-intro.html (2019. szeptember)

[6] Arduino, Intel Galileo, https://www.arduino.cc/en/ArduinoCertified/IntelGalileo (2019. szeptember)

[7] Raspberry Pi: Ubuntu 'classic' telepítés, https://wiki.ubuntu.com/ARM/RaspberryPi (2019. szeptember)

[8] The Broadcast Audio Bridge, The Difference Between Line and Mic Level Audio https://www.thebroadcastbridge.com/content/entry/7578/the-difference-between-line-and-mic-level-audio (2019. október)

[9] Real-time system scheduling, N. Audsley, A. Burns, http://beru.univ-brest.fr/~singhoff/cheddar/publications/audsley95.pdf, (2019. szeptember)

[10] Keet Malin Sugathadasa, An Introduction to Kernels. The Heart of Computing Devices., https://keetmalin.wixsite.com/keetmalin/single-post/2017/08/24/An-Introduction-to-Kernels-The-Heart-of-Computing-Devices (2019. szeptember)

[11] The Linux Information Project, Context Switch Definition, http://www.linfo.org/context_switch.html (2019. szeptember)

[12] The Raspberry Pi Foundation, Kernel Building, https://www.raspberrypi.org/documentation/linux/kernel/building.md (2019. szeptember)

[13] Stackoverflow, Instal RT Linux patch for Ubuntu https://stackoverflow.com/questions/51669724/install-rt-linux-patch-for-ubuntu (2019. szeptember)

[14] InformIT, 7.5 Device Tree Blob (Flat Device Tree), http://www.informit.com/articles/article.aspx?p=1647051&seqNum=5 (2019. szeptember)

[15] Wikipedia, Hardware Performance Counter: Implementations https://en.wikipedia.org/wiki/Hardware_performance_counter#Implementations (2019. október)

[16] Denis Bakhalov, PMU counters and profiling basics https://easyperf.net/blog/2018/06/01/PMU-counters-and-profiling-basics (2019. szeptember)

[17] Brendan D. Gregg, perf Examples, http://www.brendangregg.com/perf.html (2019. szeptember)

[18] Perf Wiki, Tutorial: Linux kernel profiling with perf https://perf.wiki.kernel.org/index.php/Tutorial (2019. szeptember)

[19] Red Hat Customer Portal, 3.13. ISOLATING CPUS USING TUNED-PROFILES-REALTIME https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_for_real_time/7/html/tuning_guide/isolating_cpus_using_tuned-profiles-realtime (2019. szeptember)

[20] A Random Walk Down Tech Street, Easy getopt for a BASH script https://dustymabe.com/2013/05/17/easy-getopt-for-a-bash-script/ (2019. október)

[21] Appfolio Engineering, What About Warmup? http://engineering.appfolio.com/appfolio-engineering/2017/5/2/what-about-warmup (2019. szeptember)

[22] Unix & Linux Stack Exchange, How do you empty the buffers and cache on a Linux system? https://unix.stackexchange.com/questions/87908/how-do-you-empty-the-buffers-and-cache-on-a-linux-system (2019. szeptember)

[23] Arm, Arm Developer, Neon, https://developer.arm.com/architectures/instruction-sets/simd-isas/neon (2019. szeptember)

[24] Arm Developer, NEON Programmer's Guide: Version 1.0 https://developer.arm.com/architectures/cpu-architecture/m-profile/docs/den0018/latest/neon-programmers-guide-version-10 (2019. szeptember)

[25] MIT OpenCourseWare Discrete-Time Signal Processing https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-341-discrete-time-signal-processing-fall-2005/index.htm (2019. október)

[26] Méréselmélet, Dr. Huba Antal, Dr. Lipovszki György http://www.mogi.bme.hu/TAMOP/mereselmelet/ (2019. október)

[27] GCC Optimize Options, https://gcc.gnu.org/onlinedocs/gcc-3.4.4/gcc/Optimize-Options.html (2019. október)

[28] Linux Journal, Introduction to Sound Programming with ALSA, https://www.linuxjournal.com/article/6735 (2019. október)

Függelék

Patch a perf eszközhöz: tools/perf/builtin-stat.c

```
1787c1787,1792
<         fprintf(output, " %17.9f seconds time elapsed", avg);
---
>         if (csv_output) {
>              fprintf(output, "%17.9f,,time elapsed,", avg);
>         }
>         else {
>              fprintf(output, " %17.9f seconds time elapsed", avg);
>         }
1794,1795c1799,1806
<              fprintf(output, " %17.9f seconds user\n", ru_utime);
<              fprintf(output, " %17.9f seconds sys\n", ru_stime);
---
>              if (csv_output) {
>                   fprintf(output,     "%17.9f,,seconds    user,\n",
ru_utime);
>                   fprintf(output,     "%17.9f,,seconds     sys,\n",
ru_stime);
>              }
>              else {
>                   fprintf(output,   "   %17.9f   seconds   user\n",
ru_utime);
>                   fprintf(output,   "   %17.9f   seconds   sys\n",
ru_stime);
>              }
1893c1904
<    if (!interval && !csv_output)
---
>    if (!interval)
```

Patch a Neon fordításához a Rust „core_arch crate”-hez: core_arch-0.1.5/src/arm/neon.rs

```
61c61
<     pub struct float32x4_t(f32, f32, f32, f32);
---
>     pub struct float32x4_t(pub f32, pub f32, pub f32, pub f32);

3d2
< #![rustfmt::skip]
```