Használati módokra optimalizált ütemezési stratégiák

Vass Dávid Attila

Miskolci Egyetem, Alkalmazott Matematikai Intézeti Tanszék

2021. június 16.

Bevezetés

- A mai világban nehezen elképzelhetőnek tűnnek azok a számítógépek, amik egyszerre csak egyetlen folyamatot képesek futtatni.
- Egy multi tasking rendszer már alapvető követelménynek tekinthető manapság az általános célú operációs rendszerek között.
- A kernel egyik fő feladata a CPU kiosztása, egy processz számára. El kell döntenie, hogy melyik futásra kész állapotú processz kapja meg a CPU-t.
- Processz ütemezőnek hívják, a kernelnek azt a részét, ami ennek az eldöntésével foglalkozik.
- Az elkészített alkalmazás célja, hogy a segítségével az adott felhasználási módhoz lehessen igazítani az ütemezési stratégiát. (Hasonlóan, mint a `tuna` és `tuned` esetében.)

Ütemező optimalizálási stratégiák

A dolgozat elejében néhány elterjedt ütemezési stratégiát is bemutatok, mint például az $\mathcal{O}(n)$, $\mathcal{O}(1)$ és a Completely Fair Scheduler-t. A jelenlegi Linux kernel ütemező, a Completely Fair Scheduler és ennek az optimalizálásra kerestem megoldásokat. Annak érdekében, hogy ezt megvalósítsam, módosításokat kell végezzek az ütemezőn, amit megtehetek többféleképpen például:

Ütemező optimalizálási stratégiák

A dolgozat elejében néhány elterjedt ütemezési stratégiát is bemutatok, mint például az $\mathcal{O}(n)$, $\mathcal{O}(1)$ és a Completely Fair Scheduler-t. A jelenlegi Linux kernel ütemező, a Completely Fair Scheduler és ennek az optimalizálásra kerestem megoldásokat. Annak érdekében, hogy ezt megvalósítsam, módosításokat kell végezzek az ütemezőn, amit megtehetek többféleképpen például:

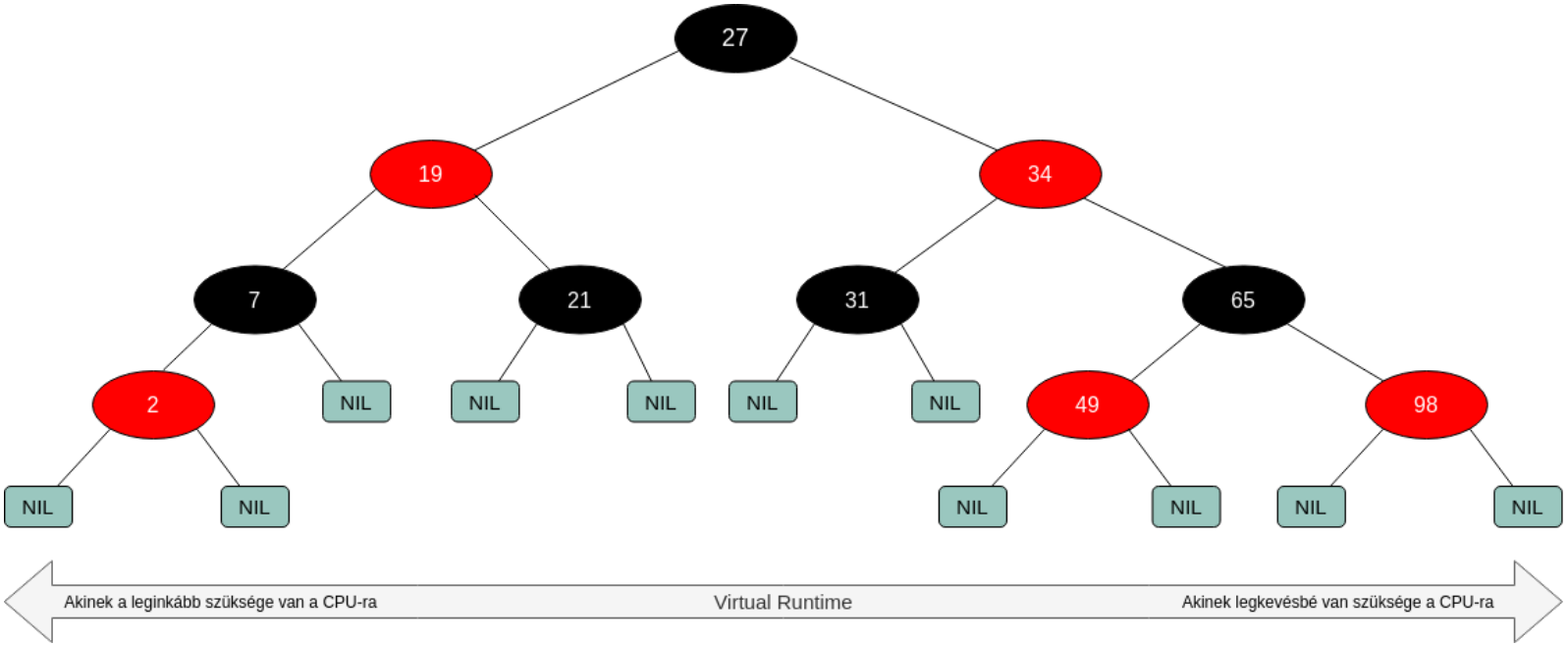
- Módosítom konkrétan a forráskódot.

Ütemező optimalizálási stratégiák

A dolgozat elejében néhány elterjedt ütemezési stratégiát is bemutatok, mint például az $\mathcal{O}(n)$, $\mathcal{O}(1)$ és a Completely Fair Scheduler-t. A jelenlegi Linux kernel ütemező, a Completely Fair Scheduler és ennek az optimalizálásra kerestem megoldásokat. Annak érdekében, hogy ezt megvalósítsam, módosításokat kell végezzek az ütemezőn, amit megtehetek többféleképpen például:

- Módosítom konkrétan a forráskódot.
- Az ütemezőhöz kapcsolódó kernel változókat módosítom.

Processz megválasztása a CFS ütemezőben



Ütemezőhöz kapcsolódó kernel változók

Az ütemezőhöz és még sok más kernel komponenshez tartozó változót, megtalálhatunk a `/proc/sys/kernel/` jegyzékben.

- Fontos, hogy ezek a változók futásidőben is módosíthatók.
- Minden változóhoz tartozik egy intervallum, amin belül tetszőlegesen változtatható az értéke.

Ezek értékeit beállítjuk többféle módon.

- Felülírjuk konkrétan a fájlban az értéket.
- Sysctl interface segítségével.

Sysctl interface

A *sysctl* interface segítségével lekérdezhetem, módosíthatom a kernel változók aktuális értékeikeit és figyelmeztet arra is, hogy az adott változó értékét ne állítsam az intervallum végpontjain túllépő értékekre. A konkrét ütemezőhöz kapcsolódó változók amiket módosítottam, a következők.

- `sched_latency_ns`
- `sched_min_granularity_ns`
- `sched_wakeup_granularity_ns`
- `sched_tunable_scaling`

Ezek mellett még két változón fogok módosításokat végezni, a processz prioritásán és a `vm.swappiness` kernel változón.

Tesztelési környezet

Az említett változókat tesztelnek kellett, hogy kiderítsem a változtatásuk milyen hatással van a rendszerre. Ehhez megpróbáltam különböző benchmark típusokat keresni, amivel különböző felhasználási módokat próbáltam szimulálni. Erre a célra a Phoronix Test Suite programot választottam, mivel számos teszt program elérhető ezen keresztül.

- Ebizzy
- PI
- Fs-Mark
- GlMark2
- Ctx-Clock
- Stream

Parameter-test program

A változókat intervallumaik szerint négy részre szedtem szét és az összes lehetséges beállítással futtattam öt mintát, minden teszt típussal. Ennek az automatizálására készítettem a *Parameter-test* programot, ami C programozási nyelv felhasználásval készült. A program feladatai:

Parameter-test program

A változókat intervallumaik szerint négy részre szedtem szét és az összes lehetséges beállítással futtattam öt mintát, minden teszt típussal. Ennek az automatizálására készítettem a *Parameter-test* programot, ami C programozási nyelv felhasználásval készült. A program feladatai:

- Előkészíteni a Phoronix Test Suite programot batch tesztelési módra.

Parameter-test program

A változókat intervallumaik szerint négy részre szedtem szét és az összes lehetséges beállítással futtattam öt mintát, minden teszt típussal. Ennek az automatizálására készítettem a *Parameter-test* programot, ami C programozási nyelv felhasználásval készült. A program feladatai:

- Előkészíteni a Phoronix Test Suite programot batch tesztelési módra.
- A kernel változók és prioritás értékeit módosítania kell.

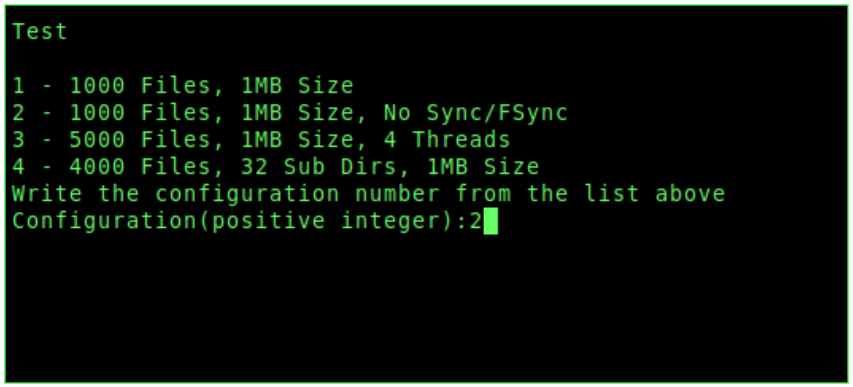
Parameter-test program

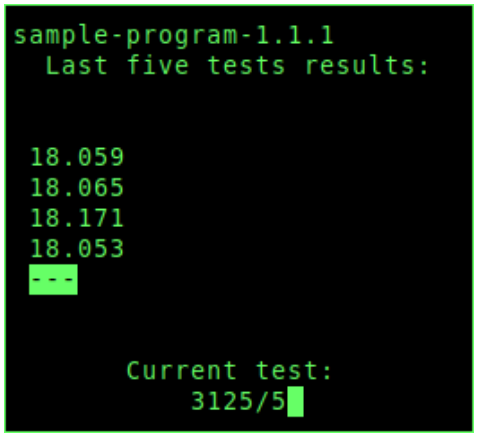
A változókat intervallumaik szerint négy részre szedtem szét és az összes lehetséges beállítással futtattam öt mintát, minden teszt típussal. Ennek az automatizálására készítettem a *Parameter-test* programot, ami C programozási nyelv felhasználásval készült. A program feladatai:

- Előkészíteni a Phoronix Test Suite programot batch tesztelési módra.
- A kernel változók és prioritás értékeit módosítania kell.
- Amikor végzett a benchmark egy adott mintaszámmal, a kapott értéket le kell mentenie egy fájlba.

Parameter-test program

- A programhoz készítettem egy ncurses-es menüt amivel kiválaszthatjuk a tesztet és futás közben láthatjuk, hogy mennyi teszt van még hátra.
- Maga a teszt egy JSON fájt készít, amibe beleírja az épp aktuális változó értékekkel elért értékekeket.
- A kimeneti fájl feldolgozását Python segítségével végeztem.





SchedulerTuneML

A *SchedulerTuneML* program Jupyter notebook-ban készült. A gépi tanulás eszközkészletével oldottam meg a paraméterek hangolását.

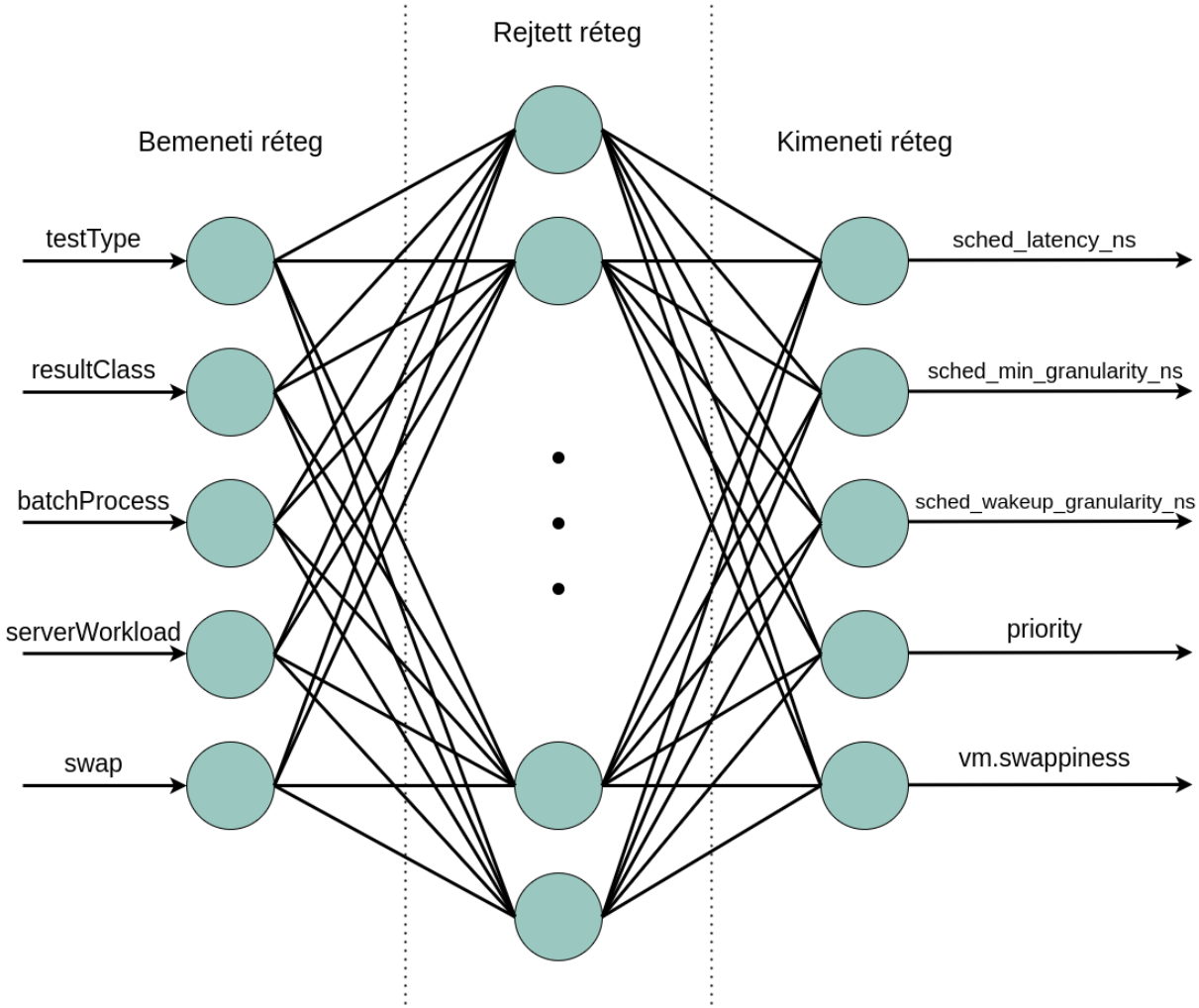
Feladai:

- A *parameter-test* program által készített JSON fájlokat feldolgozza.
- A fájlokból kinyert adatokat elemezze.
- Készítse el az adathalmazt.
- Miután a gépi tanulás befejeződött, tegyen javaslatokat a kernel változók értékeinek módosítására, az adott felhasználási módhoz viszonyítva.

SchedulerTuneML

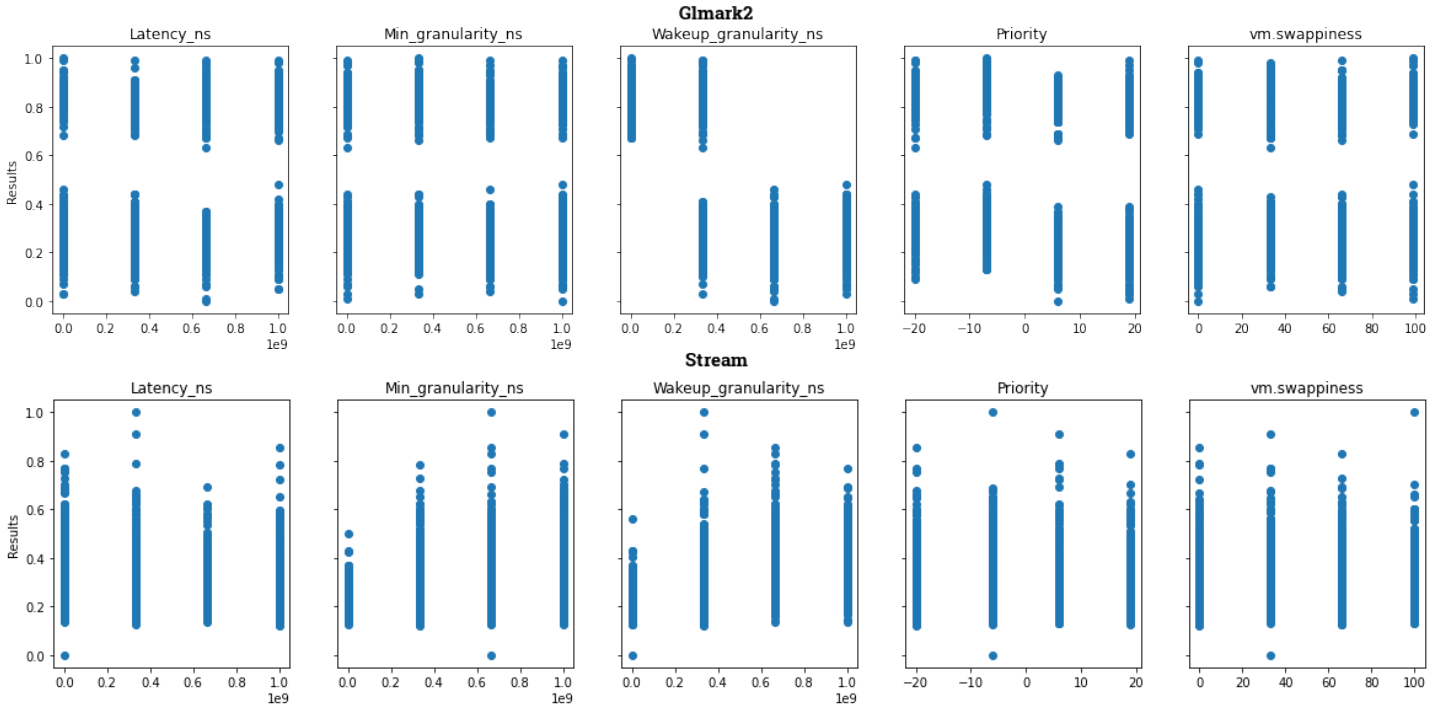
- A model öt bemenettel, illetve öt kimenettel rendelkezik, és egy rejtett réteget választottam a topológia kialakításához.
- Mivel a kernel változók intervallumait négy részre szedtem szét a teszteléseknél, így egy klasszifikációs algoritmust választottam a model betanítására.

SchedulerTuneML – A mesterséges neurális háló felépítése

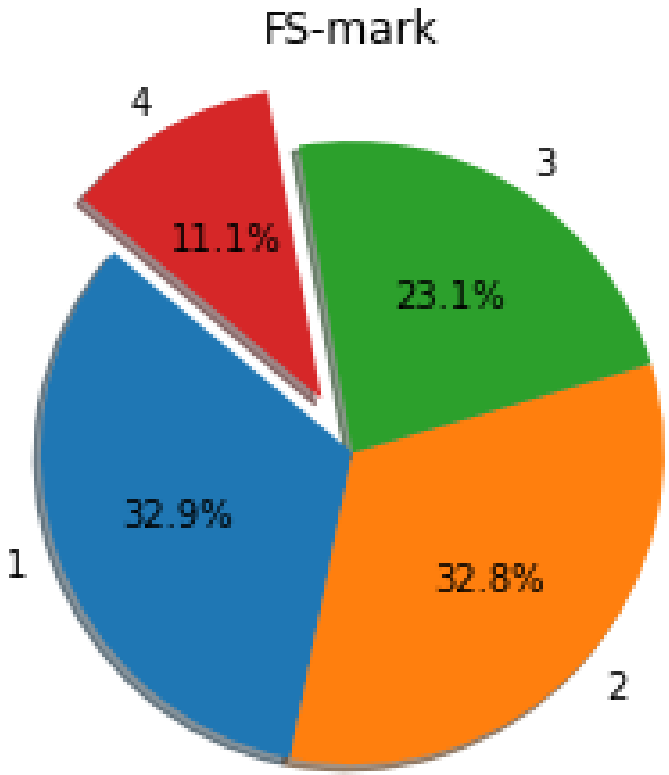


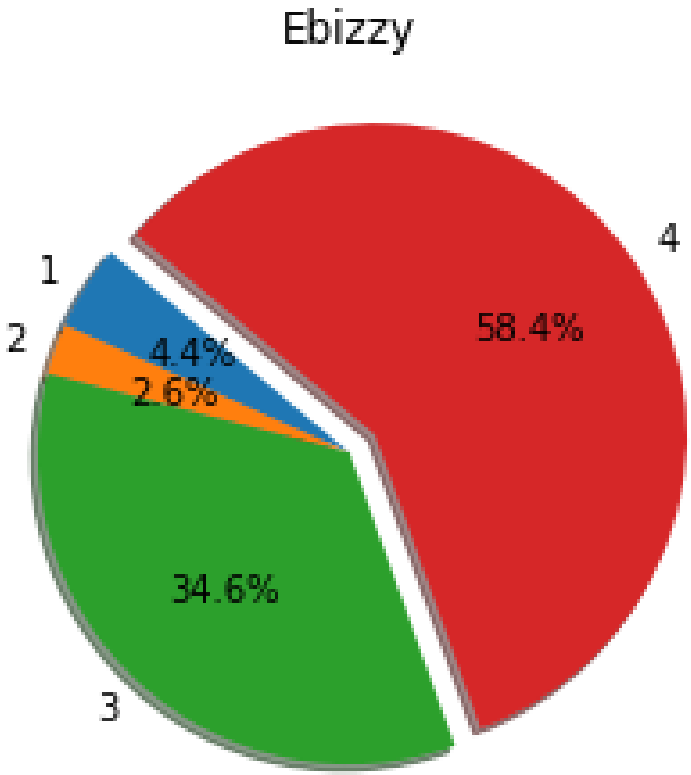
Teszt eredmények

Minden teszthez készítettem összesítő ábrákat, külön az elért eredményekhez és a paraméterekhez. Ezeken megfigyelhető az eredmények eloszlása és hogy mely változók bizonyultak hasznosabbnak a tesztek során.



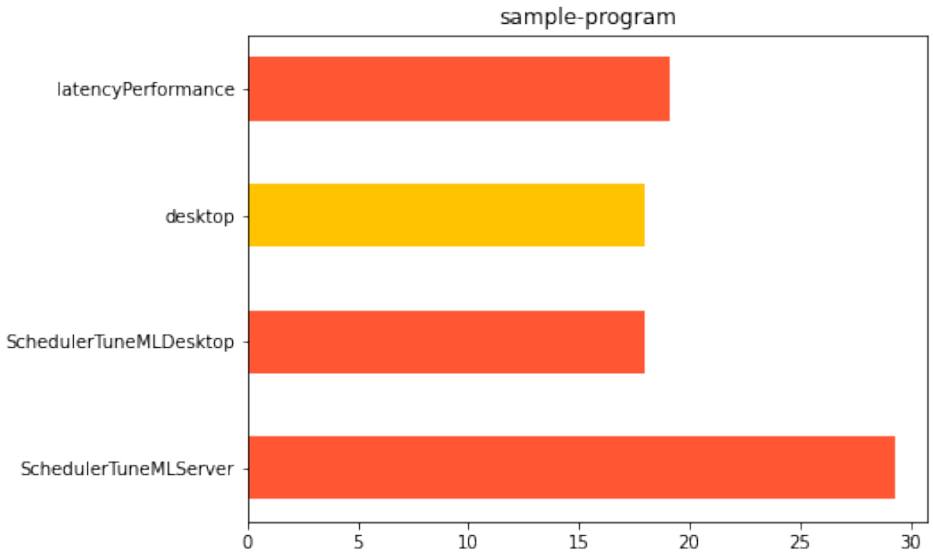
Teszt eredmények

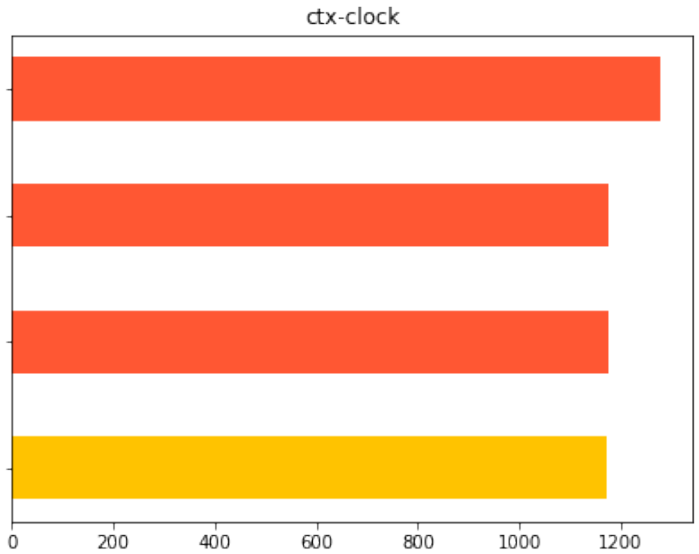




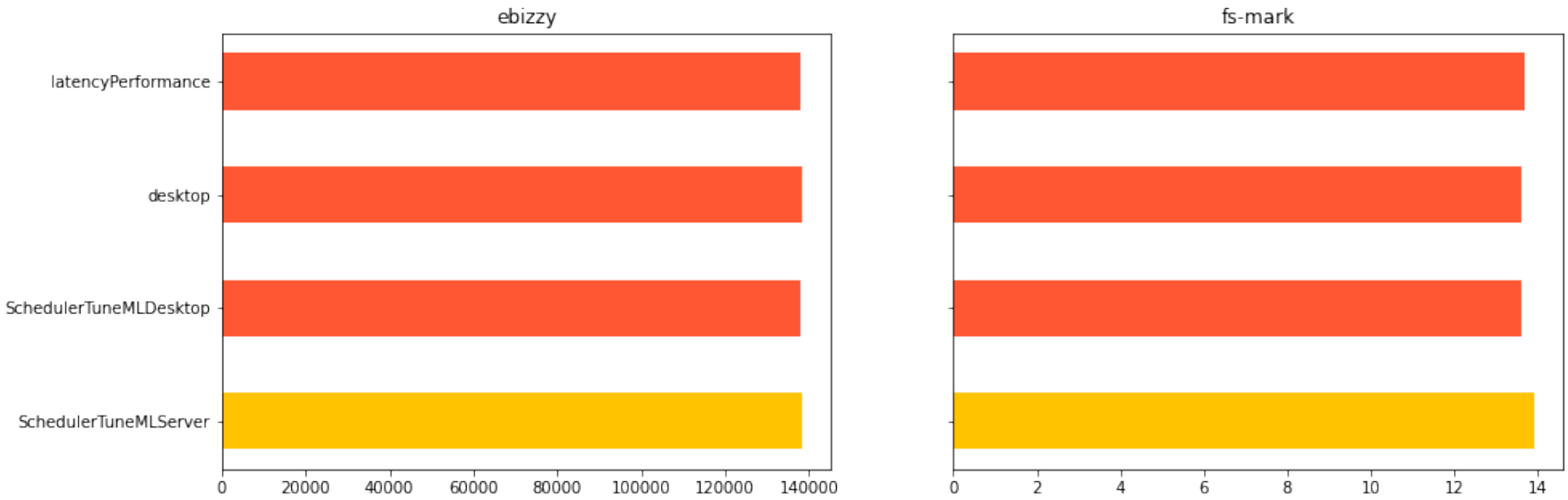
Teszt eredmények

A Red Hat által készített Tuned program, egy rendszer hangoló szolgáltatás amely elérhető GNU/Linux-on.





Teszt eredmények



Összegzés

- Áttekintésre került a Linux kernel ütemezője, annak különféle algoritmusai.
- A felhasználási módokhoz választottam egy Phoronix benchmark-ot.
- Elkészült egy ncurses alapú interfész a benchmark-ok futtatásához.
- Mesterséges neurális háló segítségével sikerült becslést adni az adott benchmark-nak megfelelő felhasználási mód optimális paramétereire.
- Mérésekkel sikerült alátámasztani az elkészített módszer helyességét.

Felhasznált (fontosabb) irodalmak

- Robert Love: Linux Kernel Development, Addison-Wesley, 2010.
- Chester Rebeiro: CPU scheduling, 2020.
- Red Hat: How to start a process with the deadline scheduler, 2019.
- Phoronix Media. Phoronix test suite, 2021.
- Mohamad H Hassoun et al. Fundamentals of artificial neural networks. MIT press, 1995.

Köszönöm a figyelmet!