



Eötvös Loránd Tudományegyetem Informatikai Kar

Energiahasználat forráskód szintű mérése C++-ban

Budapest, 2021

Témavezető:
Porkoláb Zoltán
Egyetemi docens, PhD

Szerző:
Szerencsi Kristóf
programtervező informatikus BSc

Tartalomjegyzék

1. Bevezetés.....	3
1.1 A szakdolgozat célja	3
1.2. Köszönetnyilvánítás.....	4
1.3 Intel RAPL.....	4
1.4 Intel RAPL hibái, limitei	4
1.5 Intel RAPL előnyei	5
1.6 Észrevételek, megjegyzések.....	6
1.7 Mérések alapja.....	6
2. Felhasználói dokumentáció	9
2.1 Rendszerkövetelmények	9
2.2 Futtatás	11
2.3 Hibaelhárítás.....	15
3. Fejlesztői dokumentáció.....	17
3.1 Tervezés	17
3.2 Implementáció.....	24
3.3 Tesztelés.....	32
4. Összegzés.....	45
Irodalomjegyzék	47
Ábrajegyzék	49

1. Bevezetés

1.1 A szakdolgozat célja

A szakdolgozat részletesen vizsgálja a különböző C++ nyelvbeli adatszerkezetek és programok energiafogyasztását. A társadalom környezetvédelmi aggályai változnak, és ezzel együtt változik az is, ahogy mind a számítógépgyártók, mind a szoftvergyártók fejlesztik termékeiket.

Bár az energiafogyasztás elemzése olyan terület, amely az elmúlt két évtizedben már elkezdődött, a szoftverfejlesztés ágazata csak nemrég hívta fel magára a figyelmet. A múltban a végrehajtási idő javítása volt a fő cél, amikor a hardver/szoftver, és így programozási nyelveket készítették, manapság ugyanakkor egyre jobban az energiafogyasztás kerül az előtérbe. Ez az energiatudatosság nem csak a jövőbeni szoftverek írásakor lesz elengedhetetlen, hanem a régi kódok optimalizálásakor is.

Egyik célja a szakdolgozatnak az, hogy összehasonlítsuk az energiafogyasztását különböző C++ nyelvbeli adatszerkezeteknek és programoknak a RAPL (Intel's Runtime Average Power Limit) könyvtár segítségével.

A RAPL egy olyan interfész, amely hozzáférést biztosít az energia és teljesítmény értékek leolvasásához egy modellspecifikus nyilvántartáson keresztül. Ezen eszközök segítségével lehetőségünk nyílik arra, hogy egy olyan szoftvert írjunk, amely automatikusan elhelyezi a megfelelő RAPL hívásokat a forráskódban, megkönnyítve így a különböző adatszerkezetek és programok energiafogyasztásának a mérését. Az így kapott mérési eredményeket felhasználva ajánlásokat tudunk tenni a felhasználóknak az energiahasználat szempontjai szerint optimális adatszerkezetek és algoritmusok kiválasztásához.

1.2. Köszönetnyilvánítás

Szeretnék köszönetet mondani Porkoláb Zoltán témavezetőmnek, aki mindig elérhető volt, és rengeteget segített a szakdolgozat írása közben. Bármilyen probléma felmerült, egyből ott volt hogy segítsen.

Köszönöm szépen Tanár Úr!

1.3 Intel RAPL

Running Average Power Limit (RAPL) egy interfész, amit eredetileg az Intel tervezett, hogy chip-szintű energia menedzsmentet tegyen lehetővé. A mai Intel architektúrák széles körben támogatják (Xeon szerver-szintű processzorok, valamint az i5 és i7-es CPU-k). A RAPL-t támogató architektúrák monitorozni tudják a teljesítmény számlálókat egy gépben, valamint meg tudják becsülni az energia fogyasztást úgy, hogy a becsléseket a Model-Specific Register-ben (MSR) tárolják. Az MSR-t az operációs rendszer is el tudja érni az MSR kernel modul segítségével Linuxban.

A RAPL teljesítmény számlálók lényegében 32-bites regiszterek, amik a processzor bootolásától számított idő alatti energia fogyasztását adják meg. Ezek a számlálók 1ms-ént frissülnek. Ez a fogyasztás modell-specifikus egységek többszöröseként van számolva, például: Sandy Bridge 15.3 μ J-t használ, míg a Haswell és a Skylake modellek 61 μ J-t. Néhány modellnél, mint például a Haswell-EP-nél, a DRAM-nál használt egységek eltérőek lehetnek a CPU-nál használt egységektől.

1.4 Intel RAPL hibái, limitjei

Figyelembe kell vennünk egy fontos, azonban szerencsére elég ritka esetet: regiszter túlcsordulás (register overflow). A számlálók 32 bitre vannak limitálva, azonban az MSR-ek 64

bitesek. A túlcsoordulási időt a következőképpen tudjuk kiszámolni:

$$\frac{2^{32} \times E}{P}$$

Ahol E = a modell specifikus egység (esetünkben 61.04μJ) és P a processzor fogyasztása.

Továbbá a RAPL nem támogatja a CPU magok külön-külön való mérését. A Cores csak az adott packagen belüli összes mag fogyasztását adja vissza. Minden egyes magnak saját MSR-je van, viszont a modell-specifikus egység az összesnél ugyanazt az értéket veszi fel.

Egy másik limit az is, hogy az 1ms-os számláló frissítési időt sem lehet változtatni.

1.5 Intel RAPL előnyei

Az Intel RAPL pontossága annak ellenére is magas, hogy számlálókat használ, habár a pontosság modelltől modellre változhat.

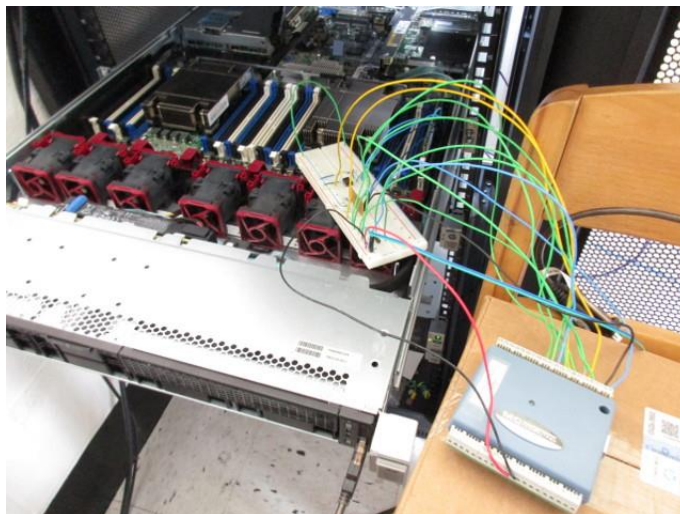
Mivel a RAPL számítások a hardverben vannak implementálva, ezért nincs szükség komplex számításokra a szoftverben.

További pozitívum, hogy egy egyszerű számítógépen kívül, nincs további szükség különböző berendezésekre, ami a RAPL-t nagyon olcsóvá teszi az energiamérésekhez.

Habár az 1.4 szekcióban megemlítettük, hogy az 1ms-os (1000Hz) számláló frissítési idejét nem tudjuk megváltoztatni, ez még mindig sokkal nagyobb érték, mint a külső teljesítménymérőké, amelyek általában másodpercenként mérik a teljesítményt.

A RAPL a processzor indításával kezdi meg a munkálatokat, így nincs szükség a beállítására, nagyon egyszerű a használata. Mivel a RAPL mindig fut, ezért a számlálók leolvasása csak nagyon kevés többletköltséget jelentenek.^[1]

¹ [\(PDF\) RAPL in Action: Experiences in Using RAPL for Power Measurements \(researchgate.net\)](#)



1.1 ábra. Példa egy külső teljesítménymérőre ^[2]

1.6 Észrevételek, megjegyzések

Természetesen az én rendszeremen mért eredmények nem garantálják azt, hogy más hardverekkel rendelkező gépeken is ugyanezek az eredmények jönnek ki. Mivel a mai felhasználóknak szánt számítógépek nem úgy lettek kialakítva, hogy azok a saját fogyasztásukat túlpontosan vissza tudják adni, valamint a nemdeterminisztikus tulajdonságuknak köszönhetően egy adott rendszeren belül is a néhány tizedes változások a fogyasztást illetően természetesnek vehetőek.

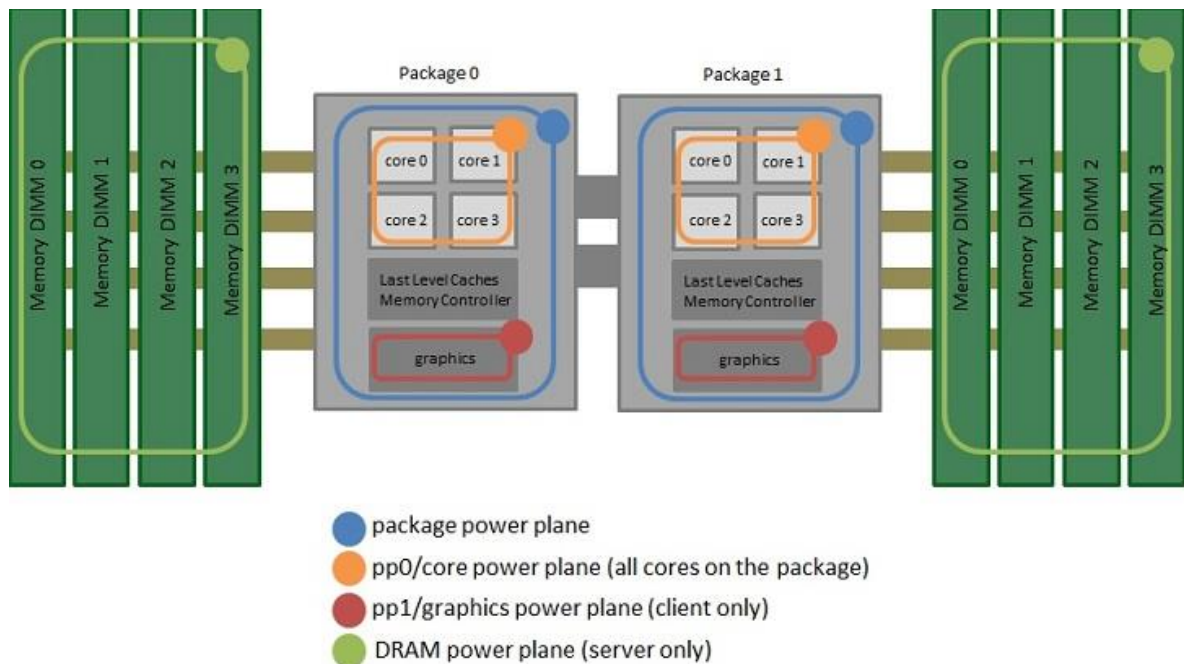
A cél az, hogy az arányokat megtartsuk: ha egy adott rendszeren belül egy M művelet F fogyasztást produkál, egy N művelet pedig $4F$ fogyasztást, akkor ezek az arányok lehetőleg (kisebb eltéréssel) maradjanak meg más rendszereken is.

1.7 Mérések alapja

Lényegében három dolgot mérünk a szakdolgozat során: a Cores (magok), a Package, valamint a RAM energiafogyasztását.^[3]

² [DRAM RAPL Validation \(maine.edu\)](https://maine.edu)

³ [Energy measurements in Linux | chih's blog](#)



1.2 ábra. Mérendő tartományok

- Egy processzornak egy vagy több package van. Ezek a processzor részei, amit az Inteltől veszünk. Kliens processzoroknak (pl: i3/i5/i7) általában egy package-ük van, míg a szerver processzoroknak (pl: Xeon) általában 2 vagy több.
- Minden egyes package-ben több cores (mag) van.
- A cores-on (narancssal jelölt) kívüli területet nevezzük uncore-nak. Ebbe a részbe tartozik például a Last Level Cache (esetemben L3 Cache), memória kontrollerek,

valamint ha elérhető, akkor az integrált GPU.

- A RAM egy külön egység a CPU-tól.

A következő kapcsolat fent áll: $PP0 + PP1 \leq \text{Package}$.

```
kristof@kristof-Lenovo-Y520-15IKBN:~/Documents/test$ ./intel-rapl -p  
RAPL read -- use -s for sysfs, -p for perf_event, -m for msr  
Found Kaby Lake Processor type  
  0 (0), 1 (0), 2 (0), 3 (0),  
  Detected 4 cores in 1 packages
```

1.3 ábra. A programunk futás közben automatikusan megmondja a processzor típusát, valamint, hogy mennyi package-el, illetve maggal rendelkezünk.

2. Felhasználói dokumentáció

2.1 Rendszerkövetelmények

Az első dolog amire szükségünk lesz az egy Intel processzor (**SandyBridge vagy későbbi**) valamint egy Linux (**3.13 vagy későbbi verzió; ajánlott: Ubuntu 20.04.2 LTS**) kernel. Az alábbi táblázat megmutatja, hogy mely processzor modellek mely funkciókat támogatják:^[4]

Name	package	PP0	PP1	RAM	powercap	perf_event
Sandybridge	Y	Y	Y	N	3.13 (2d281d8196)	3.14 (4788e5b4b23)
Sandy Bridge EP	Y	Y	N	Y	3.13 (2d281d8196)	3.14 (4788e5b4b23)
Ivy Bridge	Y	Y	Y	N	3.13 (2d281d8196)	3.14 (4788e5b4b23)
Ivy Bridge EP ("Ivy Town")	Y	Y	N	Y	no	3.14 (4788e5b4b23)
Haswell	Y	Y	Y	Y	3.16 (a97ac35b5d9)	3.14 (4788e5b4b23)
Haswell ULT	Y	Y	Y	Y	3.13 (2d281d8196)	3.14 (7fd565e27547)
Haswell GT3E	Y	Y	Y	Y	4.6 (462d8083f)	4.6 (e1089602a3bf)
Haswell EP	Y	?	N	Y	3.17 (64c7569c065)	4.1 (64552396010)
Broadwell	Y	Y	Y	Y	yes	
Broadwell-H GT3E	Y	Y	Y	Y	4.3 (4e0bec9e8)	4.6 (7b0fd569303)
Broadwell-DE	Y	Y	Y	Y	3.19 (d72be771c5d)	4.7 (31b84310c79)
Broadwell EP	Y	?	N	Y	4.1 (34dfa36c04c)	4.6 (7b0fd569303)
Skylake Mobile	Y	Y	Y	Y	4.1 (5fa0fa4b01)	4.7 (dcee75b3b7f02)
Skylake Desktop H/S	Y	Y	Y	Y	4.3 (2cac1f70)	4.7 (dcee75b3b7f02)
SkylakeServer	Y	Y	N	Y	4.8???	4.8 (348c5ac6c7dc11)

⁴ [Linux support for Power Measurement Interfaces \(maine.edu\)](https://www.maine.edu/~linux/)

Kabylake	Y	Y	Y	Y	4.7 (6c51cc0203)	4.11 (f2029b1e47)
Cannonlake	Y	Y	Y	Y	4.17 (?)	4.17 (490d03e83da2)
Knights Landing	Y	N	N	Y	4.2 (6f066d4d2)	4.6 (4d120c535d6)
Knights Mill	Y	N	N	Y	()	4.9 (36c4b6c14d20)
Atom Goldmont	Y	Y	Y	Y	4.4 (89e7b2553a)	4.9 (2668c6195685)
Atom Denverton	Y	Y	Y	Y	()	4.14 (450a97893559354)
Atom Gemini	Y	Y	Y	Y	()	4.14 (450a97893559
Atom Airmont /	?	?	?	?	3.19 (74af752e489	no
Atom Tangier /	?	?	?	?	3.19 (74af752e48	no
Atom Moorefield Annidale	?	?	?	?	3.19 (74af752e4895)	no
Atom Silvermon Valleyview	?	?	?	?	3.13 (ed93b71492d)	no

2.1 táblázat. Processzor modell funkciók

A méréseket a következő paraméterekkel végeztem el:

Processzor:	Intel® Core™ i5-7300HQ CPU @ 2.50GHz
RAM:	8GiB SODIMM DDR4 Synchronous Unbuffered (Unregistered) 2400MHz (0,4 ns)
L1 Cache:	256KiB
L2 Cache:	1MiB
L3 Cache:	6MiB
GPU:	GP107M [GeForce GTX 1050 Mobile]
OS:	Ubuntu 20.04.2 LTS
Kernel:	Linux 5.8.0-48-generic
Architecture:	X86-64

2.2 táblázat. Saját gép paramétere

A fenti adatokat az alábbi parancsokkal tudjuk lekérdezni:

```
1. $ sudo lshw -short
2. $ hostnamectl
```

2.2 Futtatás

Nyissunk egy terminált, és egy tetszőleges mappába töltjük le a forráskódot:

```
1. $ git clone https://github.com/sz-kristof/intel-rapl
2. $ cd intel-rapl
```

Győződjünk meg arról, hogy rendelkezésünkre áll a g++ fordító.

```
1. $ g++ --version
```

Amennyiben még nem installáltuk, a következő paranccsal megtehetjük:

```
1. $ sudo apt install build-essential
```

Jelenleg három módszer van arra, hogy a RAPL eredményeket le tudjuk olvasni egy Linux kernel segítségével:

1. A fájlokat a `/sys/class/powercap/intel-rapl/intel-rapl:0` alatt olvassuk le a powercap interfész segítségével. Ennél a módszernél nincs szükségünk további engedélyekre, és a Linux 3.13-as verzióval lett bemutatva.
2. Használhatjuk a `perf_event` interfészt, ha Linux 3.14 vagy újabb verziónk van. Ehhez root-ra van szükségünk, vagy arra hogy egy *paranoid* értéke kisebb legyen mint 1: a `/proc/sys/kernel/perf_event_paranoid` < 1 teljesülnie kell. Ennek ellenőrzése egyszerű: nyissunk egy új terminált majd írjuk be a következő parancsot:

```
a. $ sudo vim /proc/sys/kernel/perf_event_paranoid
```

Amennyiben az értékünk kisebb mint 1, a `:q` paranccsal kiléphetünk és további dolgunk nincs. Amennyiben az értékünk nagyobb vagy egyenlő mint 1, nyomjuk meg az `i` betűt, írjuk át a bent lévő értéket 0-ra, majd nyomjunk egy `Esc` gombot, és a `:wq!` paranccsal mentjük a fájlt.

A szakdolgozat alatt is ezt a módszert használtam.

3. “Nyers” hozzáférés az MSR-ekhez a `/dev/msr` alatt. Ehhez root-ra van

szükségünk, és arra, hogy a `/dev/cpu/??/msr` driver engedélyezve legyen, valamint read jogosultságra is szükségünk lesz. Érdemes lehet lefuttatni a `modprobe msr` parancsot, ha nem működik.

Ezek után a meglévő `.cpp` kódunkat fordíthatjuk a

```
1. g++ -o <outputname> <filename>.cpp
```

parancs segítségével. Ha ez megvan, futtassuk a lefordított állományt a

```
1. ./<outputname> -p
```

parancssal. A `-p` kapcsoló segítségével mondjuk meg hogy a `perf_event` interfészt akarjuk használni.

Ha minden jól ment, akkor a futtatás után a következőt kell látnunk:

```
RAPL read -- use -s for sysfs, -p for perf_event, -m for msr

Found Kaby Lake Processor type
      0 (0), 1 (0), 2 (0), 3 (0),
      Detected 4 cores in 1 packages

---MENU---
Choose the container you want to use:
1.) vector
2.) deque
3.) set
4.) multiset
5.) unorderedset
6.) list
---OR---
Measure basic consumption (press x)
Type here: 
```

2.1 ábra. Konténerek menüpont

Itt egy menüt láthatunk, amiben kiválaszthatjuk, hogy mely konténert szeretnénk lemérni. Írhatjuk a megfelelő számot vagy a nevet is, ha elgépelünk valamit a program újból megkérdezi, hogy mi a választásunk. Ha kiválasztottuk a megfelelő konténert, akkor a következőt láthatjuk (deque esetében):

```
Current container: deque
Choose the method you want to measure:
1.) insert
2.) push_back
3.) push_front
4.) emplace_back
5.) emplace_front
Type here: █
```

2.2 ábra. Függvények menüpont

Itt pedig a kiválasztott konténer függvényei vannak felsorolva, amelyekre a méréseket elvégezhetjük. Ha ezzel is megvagyunk, akkor a program leméri az általunk kiválasztott konténer/függvény fogyasztását. Ennek az outputja a következőképpen néz ki:

```
Measuring deque emplace_front performance:

Time: 139.667ms
Cores Result 1.49419 J
Package Result 1.9129 J
Ram Result 0.0635274 J

Cores values:
1.48114J 1.48218J 1.49286J 1.49524J 1.5061J 1.50763J
Package values:
1.89716J 1.90375J 1.90833J 1.91132J 1.92719J 1.92969J
RAM values:
0.0628662J 0.0634766J 0.0635986J 0.0635986J 0.0636597J 0.0639648J
Time values
139ms 139ms 140ms 140ms 140ms 140ms
```

2.3 ábra. Teljes fogyasztás mérése

```

Type here: x
Type the amount (ms) you want to measure: 139

Time: 139ms
Cores Result 0.0161438 J
Package Result 0.424845 J
Ram Result 0.0365804 J

Cores values:
0.00726318J 0.0101318J 0.0131226J 0.0133057J 0.0170898J 0.0359497J
Package values:
0.418213J 0.419006J 0.421143J 0.423523J 0.424133J 0.443054J
RAM values:
0.0359497J 0.0359497J 0.0363159J 0.0369873J 0.0370483J 0.0372314J
Time values
139ms 139ms 139ms 139ms 139ms 139ms

```

2.4 ábra. Alapfogyasztás mérése

Az eredményeket a következőképpen számítjuk ki: 10 mérést végzünk majd a 2-2 legnagyobb, illetve legkisebb méréseket elvesszük, majd a maradék 6 mérést átlagoljuk.

A képeken is ez jelenik meg.

A "Time" sor a hat mérés időátlaga. A "Cores Result" a magok átlagfogyasztását adja meg. A "Package Result" a package, a RAM pedig a RAM átlagfogyasztását adja meg.

Alatta pedig ("Cores values", "Package values" stb.) a megmaradt hat darab mérés eredményeit láthatjuk, amiből az átlagokat is számítjuk.

A teljes fogyasztás (2.3 ábra) és az alapfogyasztás (2.4 ábra) között annyi a különbség, hogy a teljes fogyasztás eredményét akkor kapjuk, amikor egy adott konténer függvényét szeretnénk mérni, viszont ebből ki kell vonnunk a számítógépünk alapfogyasztását (sleep művelet), hogy megkapjuk a tényleges energiaigényét a konténernek.

Fontos megjegyezni, hogy érdemes minden háttérben futó processzt (pl: böngésző) leállítani, és csak ezt a programot futtatni az optimális futtatási környezet eléréséhez.

2.3 Hibaelhárítás

A telepítés, illetve futtatás alatt több hiba is felléphet, ha véletlenül kifejeztünk egy lépést az előbbi szekcióból.

Például megeshet, hogy a *git* nincs telepítve, így nem tudjuk leszedni a forráskódot.

Ennek persze egyszerű a megoldása, simán csak telepítenünk kell a gitet a következő paranccsal:

```
1. sudo apt-get install git
```

A fordító esetében már megemlítettük a parancsot, amellyel telepíteni lehet, az egyetlen hiba, amit itt elkövethetünk, ha optimalizálást állítunk be egy kapcsolóval. Fontos, hogy ezt ne tegyük meg, tehát ne legyen a fordító parancsunkban egy *-Ox* (x az optimalizálás szintje) kapcsoló.

A futtatás során az egyik probléma az lehet, ha a processzorunk nem támogatja a modell-specifikus regisztereket, illetve, ha az Intel RAPL nem támogatja az adott modellt. Ilyenkor nem tudunk mást tenni, mint beszerezni egy processzort a 2.1-es táblázat alapján. Éppen ezért érdemes még a program futtatása előtt meggyőződni, hogy milyen típusú processzorunk van.

A másik hibába pedig akkor futhatunk, ha a 2.2-es fejezetben említett *perf_event_paranoid* értéke nem kisebb mint 1. Ekkor a *“The value must be 0 or lower to read system-wide RAPL values”* üzenetet kapjuk, ezt természetesen a 2.2 fejezet alapján javíthatjuk. Ha esetleg a vim nincs telepítve, akkor a következő parancsokkal telepíthetjük:

```
1. sudo apt-get update
2. sudo apt-get install vim
```

A paranoid értékének változtatása közben előfordulhat egy szinkronizációs hiba, ami a következőképpen néz ki:

```
"/proc/sys/kernel/perf_event_paranoid"  
"/proc/sys/kernel/perf_event_paranoid" E667: Fsync failed  
Press ENTER or type command to continue
```

2.5 ábra. Szinkronizációs hiba

Ekkor a következő parancsot kell használnunk:

```
1. sudo sh -c  
    'echo kernel.perf_event_paranoid=0 > /etc/sysctl.d/local.conf'
```

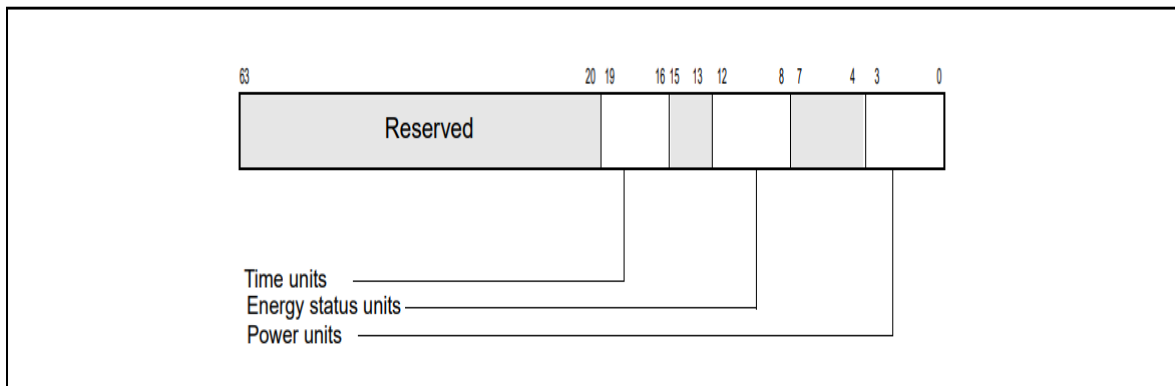
Ezek után indítsuk újra a gépünket, ha ez megvolt, akkor már az új értéknek kell szerepelni a paranoidnál.

3. Fejlesztői dokumentáció

3.1 Tervezés

A tervezés első részében megnézzük, hogy a különböző modell-specifikus regiszterek hogyan épülnek fel, azért, hogy az implementáció, illetve tesztelés során már könnyebb dolgunk legyen.

Az első ilyen regiszter amit megnézünk, az a modell-specifikus egység regisztere, amit az alábbi képen láthatunk^[5]:



3.1 ábra. MSR_RAPL_POWER_UNIT regiszter

A power unit négy bitet használ a teljesítménnyel kapcsolatos (Wattban mért) információk tárolásához, ami az $\frac{1}{2}^{\text{PU}}$ tényezőn alapul, ahol a PU változó a 3:0 bitek által van reprezentálva. Az alapértelmezett érték 0011b (3 decimális).

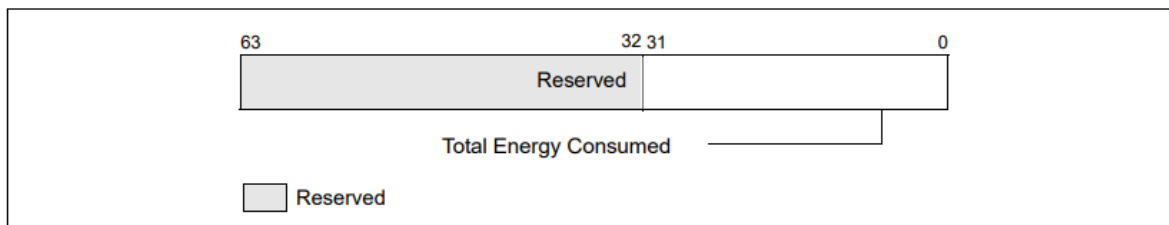
Az energy status units (modell-specifikus egység) már öt bitet használ az energiával kapcsolatos (Joules-ban mért) információk tárolásához, ami az $\frac{1}{2}^{\text{ESU}}$ tényezőn alapul, ahol az ESU változó a 12:8 bitek által van reprezentálva. Az alapértelmezett érték 10000b (16 decimális).

A time units pedig a regiszter frissítési idejét adja meg (másodpercben), ami az $\frac{1}{2}^{\text{TU}}$ tényezőn alapul, ahol az TU változó a 19:16 bitek által van reprezentálva. Az alapértelmezett

⁵ [Intel® 64 and IA-32 Architectures Software Developer Manual: Vol 3](#)

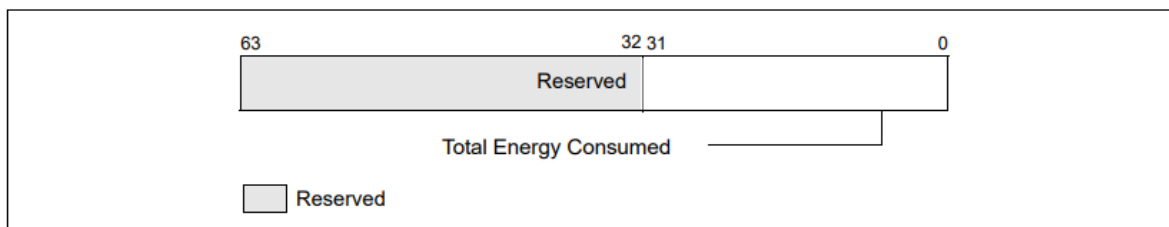
érték 1010b (10 decimális).

A következő képeken pedig láthatjuk, hogy a teljesítmény mérő regiszterek hogyan épülnek fel.



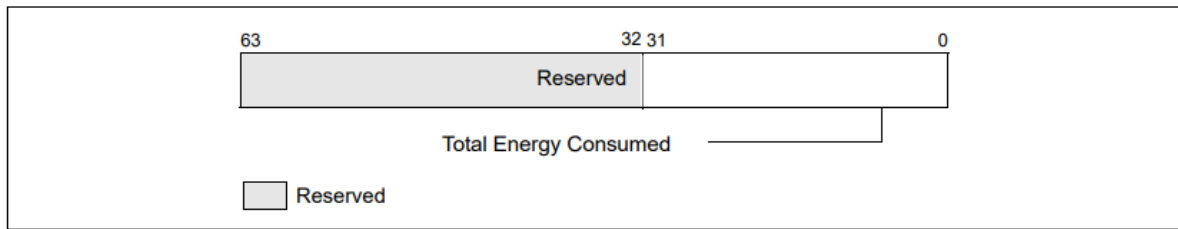
3.2 ábra. MSR_PKG_ENERGY_STATUS MSR

A Total Energy Consumed megadja, hogy mennyi energiát fogyasztott a package, amióta ez a regiszter törölve lett. Ennek az egysége az előbb megadott Energy Status Units értéke (esetünkben $61.04\mu\text{J}$).



3.3 ábra. MSR_PPO_ENERGY_STATUS MSR

A Total Energy Consumed megadja, hogy mennyi energiát fogyasztott a Cores, amióta ez a regiszter törölve lett. Ennek az egysége az előbb megadott Energy Status Units értéke (esetünkben $61.04\mu\text{J}$).



3.4 ábra. MSR_DRAM_ENERGY_STATUS MSR

A Total Energy Consumed megadja, hogy mennyi energiát fogyasztott a RAM, amióta ez a regiszter törölve lett. Ennek az egysége az előbb megadott Energy Status Units értéke (esetünkben $61.04\mu\text{J}$).

Processzorok fogyasztása

Most már ki tudjuk olvasni az MSR-ből az értékeket, viszont felmerülhet a kérdés, hogy a számítógépünk mi alapján növeli az adott MSR számlálókat?

Lényegében négy részre oszthatjuk a processzorunk teljesítményének a figyelését:

1. uncore dinamikus teljesítmény
2. core dinamikus teljesítmény
3. uncore statikus teljesítmény
4. core statikus teljesítmény

Ahol:

$$P_{\text{cpu}} = P_{\text{statikus}} + P_{\text{dinamikus}} + P_{\text{sc}}$$

A statikus teljesítmény az áramkör szivárgási teljesítménye. A tranzisztor különbözőképpen adalékolt (angolul doping) részei között mindig folyik egy kis mennyiségű áram. Ezen áramok nagysága a tranzisztor állapotától, méreteitől, fizikai tulajdonságaitól és néha a hőmérséklettől függ. A szivárgási áram összmenyisége a hőmérséklet növekedésével és a tranzisztor méretének csökkenésével általában megnő.

A dinamikus energiafogyasztás a CPU-n belüli logikai kapuk tevékenységéből ered. Amikor a

logikai kapuk kapcsolnak, energia áramlik. A CPU által fogyasztott dinamikus teljesítmény megközelítőleg arányos a CPU frekvenciájával és a CPU feszültségének négyzetével:^[6]

$$P_{\text{dinamikus}} = CV^2f$$

Ahol V a feszültség, f a frekvencia és C a kapcsolási kapacitás. (A legtöbb áramkör esetében az energiafogyasztás fő összetevője a kapcsolási teljesítmény, amely akkor kerül felhasználásra, amikor egy kapu kapcsol.)

A P_{sc} pedig a rövid-áramköri teljesítmény (short-circuit power), amely annak köszönhető, hogy amikor a különböző logikai kapuk kapcsolnak, a tranzisztorok állapotot váltanak, és mivel ez véges időt vesz igénybe, egy nagyon rövid ideig megtörténhet hogy néhány tranzisztor egyszerre vezet áramot. Az emitter és a bázis közötti közvetlen út ekkor némi rövidzárlatos teljesítményvesztést eredményez.

Konténerek felépítése

Az egyik dolog még, amit érdemes lehet megvizsgálnunk, hogy a mérendő konténerek milyen módon vannak implementálva. Ha ezeket tudjuk, akkor már nagyobb rálátásunk lesz arra, hogy milyen mérési eredményekre számíthatunk. Ez akár még a validálásban is segíthet nekünk.

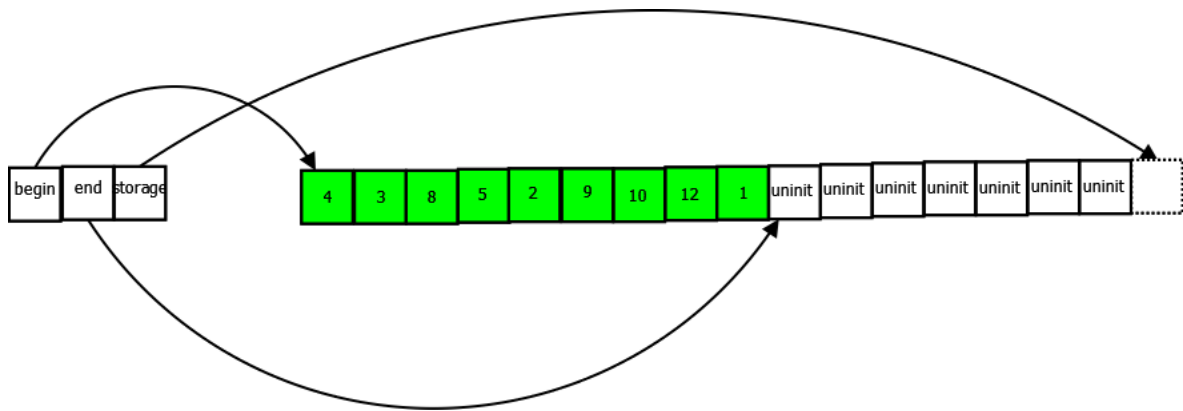
Sequence (szekvenciális) konténerek

Ebből a csoportból három darab implementációt fogunk mérni, a *vector*, a *deque*, illetve a *list*.

A szekvenciális konténerek olyan konténerosztályok, amelyek fenntartják a konténerben lévő elemek sorrendjét. A szekvencia-konténerek meghatározó jellemzője, hogy pozíció szerint választhatjuk meg, hogy hova helyezzük be az elemet.

⁶ [Processor power dissipation- Wikipedia](#)

Az egyik talán legtöbbet használt ilyen konténer a vector. A vector egy dinamikusan allokált elemekből álló tömb.



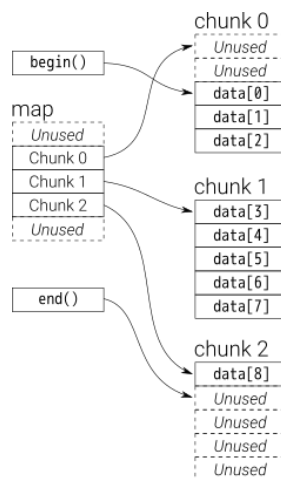
3.5 ábra. Vektor implementáció^[7]

A vektor tárolása automatikusan történik, szükség szerint bővítve és szűkítve. A vektorok általában több helyet foglalnak el, mint a statikus tömbök, mivel több memóriát rendelnek hozzá a jövőbeli növekedés kezelésére. Így egy vektornak nem kell minden egyes elem beillesztésekor újra allokálni, hanem csak akkor, amikor a további memória kimerül.

A deque némileg rekurzívan definiált: belsőleg egy fix méretű darabokból álló, kétvégű queue-t tart fenn. Minden darab egy vektor, és maga a darabok sora ("map" az alábbi ábrán) is egy vektor.^[8]

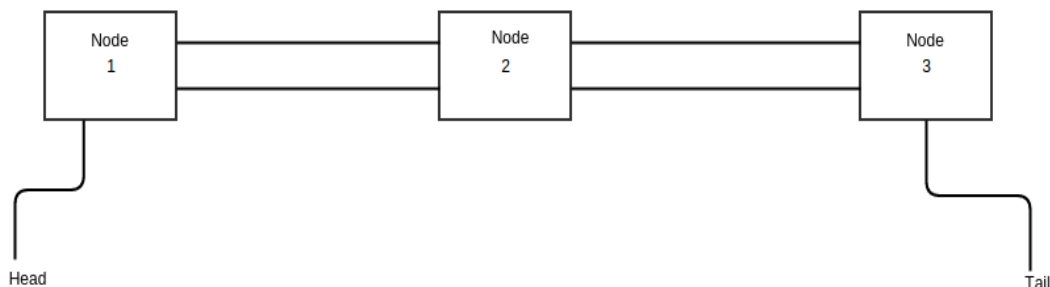
⁷ [How C++'s vector works: the gritty details « Frogatto & Friends](#)

⁸ [c++- What really is a deque in STL?- Stack Overflow](#)



3.6 ábra. Deque implementáció

A lista nem egybefüggő memóriahelyen tárolja az elemeket, azaz belsőleg egy duplán összekapcsolt listát használ:^[9]



3.7 ábra. List implementáció

Associative (asszociatív) konténerek

Az asszociatív tárolókban az elemek előre meghatározott sorrendben kerülnek beillesztésre- például növekvő sorrendben. Rendezetlen asszociatív konténerek is rendelkezésre állnak.

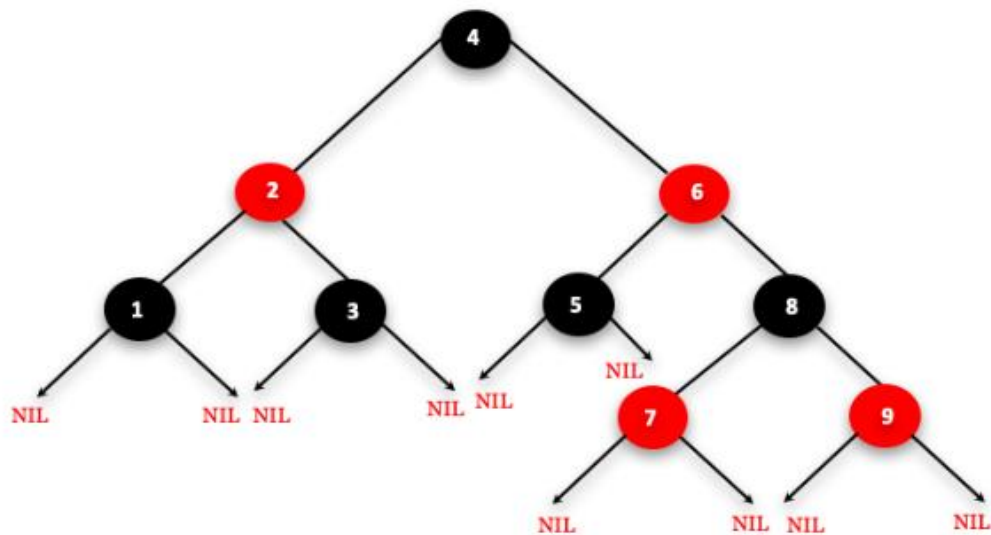
A szakdolgozat során megnézzük a *set*, *multiset*, illetve *unordered set* teljesítményét.

⁹ [Difference between Vector and List in C++ – thispointer.com](https://thispointer.com/difference-between-vector-and-list-in-c++/)

Az set egy asszociatív tároló, amely a Key típusú egyedi objektumok rendezett halmazát tartalmazza. A keresési, eltávolítási és beszúrási műveletek logaritmikus komplexitásúak. A halmazokat általában piros-fekete fákként (red-black trees) valósítják meg.

A multiset-ben a set-tel ellentétben több azonos értékű kulcs is megengedett. A keresési, beszúrási és eltávolítási műveletek itt is logaritmikus bonyolultságúak.

Az unordered set-nél belsőleg az elemek nem kerülnek sorrendbe, hanem vödrökbe rendezzük őket. Az, hogy egy elem melyik vödörbe kerül, az értékének hash-értékétől függ. Ez lehetővé teszi az egyes elemek gyors elérését, mivel a hash kiszámítása után az elem pontosan arra a vödrre utal, amelyikbe az elem került.^[10]



3.8 ábra. Piros-fekete fa implementáció^[11]

A C++ Standard 23.1.1-es szekciója^[12] egyébként azt mondja, hogy a vektort a legtöbb esetben használhatjuk, azonban ha gyakoriak a beszúrák illetve törlések a szekvencia közepéből, akkor érdemes list-et használni, ha pedig a végéből vagy elejéből, akkor inkább a deque-t használjuk.

A szekvenciális konténerek esetében egyébként a beszúrákra a garanciánk általában $O(n)$

¹⁰ [Containers library- cppreference.com](https://en.cppreference.com/containers)

¹¹ [An Introduction to Red-Black Tree | by Gild Academy — https://www.gildacademy.in/ | Medium](https://www.gildacademy.in/)

¹² [C++STANDARD-ISO/IEC14882-1998.pdf \(akkit.org\)](https://ericniebler.com/2017/02/20/cplusplus/)

és $O(1)$ között mozog, míg az asszociatív konténerekre ez már inkább $O(\log n)$.^[13]

Cache

Az egyik dolog még, amit letesztelhetünk, az az, hogy vajon a RAPL-al képesek vagyunk-e a cache-k hatását mérni a programunkban.

Megnézhetjük, hogy vajon a konténer nagyságát az L1, illetve L2-es cache méretéhez igazítva milyen eredményeket kapunk. Elvileg 256KiB illetve 1MiB után kéne látnunk egy nagyobb ugrást.

3.2 Implementáció

Egységek kiolvasása

Az egységek kiolvasása roppant érdekes művelet.

Ha beírjuk egy böngésző keresőjébe hogy `MSR_RAPL_POWER_UNIT` akkor rögtön kidob egy oldalt, ami megmondja nekünk hogy melyik fájlban van definiálva.^[14]



Defined in 2 files as a macro:

`arch/x86/include/asm/msr-index.h`, line 287 (as a macro)

`tools/arch/x86/include/asm/msr-index.h`, line 283 (as a macro)

3.9 ábra. `MSR_RAPL_POWER_UNIT` definiálásának helye

Most meg kell keresnünk az `msr-index.h` nevezetű header fájlt.

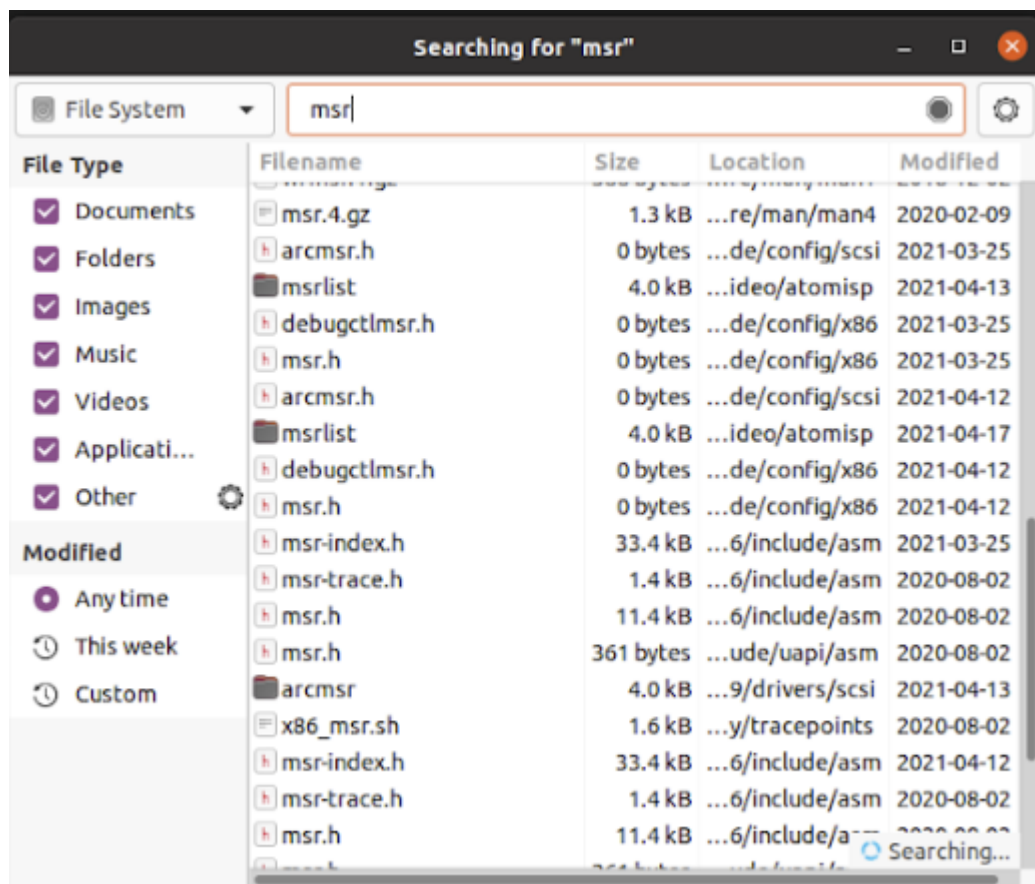
Egy egyszerű és kényelmes megoldás a **catfish** nevezetű program. Telepíthetjük a következő paranccsal:

```
1. sudo apt install catfish
```

¹³ [EECS 311: STL Containers \(northwestern.edu\)](https://www.northwestern.edu/stl/containers/)

¹⁴ [MSR_RAPL_POWER_UNIT identifier- Linux source code \(v5.8-rc1\)- Bootlin](#)

Ezek után indítsuk a programot a *catfish* paranccsal, a felugró ablakban nyomjuk meg az F9 gombot, majd bal oldalt a fájl típusoknál jelöljük ki az összeset, a keresés helyéhez jelöljük ki az egész rendszert (File System), majd a keresőbe írjuk be, hogy “msr”.



3.10 ábra. Catfish program grafikus felülete

Ha lejjebb görgetünk, már meg is találtuk a fájlt. Dupla kattintással meg is nyithatjuk a tartalmát.

A weboldal szerint a 287-es sorban lett definiálva, és valóban, a következő értéket láthatjuk:

```
1. #define MSR_RAPL_POWER_UNIT 0x00000606
```

A jobb oldalon található szám egy hexadecimális szám, ami lényegében a memória címet adja meg.

Kicsit lejjebb a különböző energia státuszokat is megtaláljuk (package, dram, cores).

Itt mérjük lényegében a processzor bootolásától mért energia fogyasztásunkat.

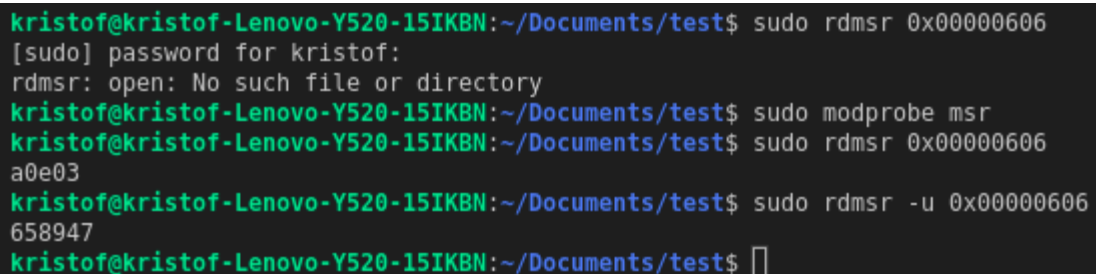
```
1. #define MSR_PKG_ENERGY_STATUS      0x00000611
2. #define MSR_DRAM_ENERGY_STATUS     0x00000619
3. #define MSR_PP0_ENERGY_STATUS      0x00000639
```

A kérdés már csak az, hogy ezekből a memória címekből hogyan tudjuk kiolvasni az értékeket? Az egyik lehetőség az *rdmsr* parancs. Ehhez viszont előbb le kell futtatnunk a *modprobe msr* parancsot, különben nem találnánk semmit.

```
1. $ sudo modprobe msr
2. $ sudo rdmsr 0x00000606
```

Az eredményt hexadecimális formában kapjuk (esetemben a0e03), viszont az *-u* kapcsolóval már decimális formában kapjuk meg.

```
1. $ sudo rdmsr -u 0x00000606
```



```
kristof@kristof-Lenovo-Y520-15IKBN:~/Documents/test$ sudo rdmsr 0x00000606
[sudo] password for kristof:
rdmsr: open: No such file or directory
kristof@kristof-Lenovo-Y520-15IKBN:~/Documents/test$ sudo modprobe msr
kristof@kristof-Lenovo-Y520-15IKBN:~/Documents/test$ sudo rdmsr 0x00000606
a0e03
kristof@kristof-Lenovo-Y520-15IKBN:~/Documents/test$ sudo rdmsr -u 0x00000606
658947
kristof@kristof-Lenovo-Y520-15IKBN:~/Documents/test$
```

3.11 ábra. Rdmsr parancs futtatása

Viszont a kódunkból kéne valahogy kiolvasnunk ezeket az értékeket. Ezt a következőképpen csinálhatjuk meg:

```

1. static long long read_msr(int fd, int which) {
2.
3.     uint64_t data;
4.
5.     if ( pread(fd, &data, sizeof data, which) != sizeof data ) {
6.         perror("rdmsr:pread");
7.         exit(127);
8.     }
9.
10.    return (long long)data;
11.}

```

Ahol *fd* a file descriptorunk, *which* a memóriacímünk (0x606), *&data* egy pointer a bufferünkhöz, *sizeof data* pedig megadja, hogy hány bájtot szeretnénk olvasni. A visszatért értéket nevezzük el *result*-nak, ekkor a PU értéke:

```

1. power_units=pow(0.5,(double)(result&0xf));

```

Az egyetlen furcsaságot a sor végén találhatjuk. Mivel csak az első négy bitet használjuk a power unit leírásához, ezért egy bitwise and operátorral vesszük az első négy bitet, ami a következőképpen néz ki:

0xf = 1111

a0e03 = 10100000111000000011

10100000111000000011 & 1111 = 00000000000000000011 vagy 0011 aminek az értéke pont 3. Végül tehát $0.5^3 = 0.125W$.

```
RAPL read -- use -s for sysfs, -p for perf_event, -m for msr
Found Kaby Lake Processor type
  0 (0), 1 (0), 2 (0), 3 (0),
  Detected 4 cores in 1 packages

Trying /dev/msr interface to gather results

Listing parameters for package #0
Power units = 0.125W
CPU Energy units = 0.00006104J
DRAM Energy units = 0.00006104J
Time units = 0.00097656s

Package thermal spec: 45.000W
Package minimum power: 0.000W
```

3.12 ábra. A programunk futtatás után kiírja az egységeket

Az energy status units lényegében a mi modell-specifikus egységünk lesz. A számolási menet itt is hasonló lesz egy kis eltéréssel. A kódunkban természetesen ugyanarról a címről olvasunk, tehát csak egy új számítást kell elvégeznünk:

```
1. cpu_energy_units=pow(0.5,(double)((result>>8)&0x1f));
```

A \gg operator C-ben a right shift operátort jelenti. Ha decimális számokat nézünk akkor a right shift operátor a következőt jelenti:

$$N=N\gg 2 \rightarrow N=N/(2^2)$$

Tehát ha például $N = 32$ (100000b) akkor:

$$N=32/(2^2)=8 \text{ (1000b)}$$

Erre azért van szükségünk mert csak a nyolcadik bittől kell nézni az értékünket.

A második különbség pedig az, hogy a bitwise operátor job oldalán nem 0xf található, hanem 0x1f, ami annak köszönhető, hogy most 4 bit helyett 5 bitet használunk.

A számolás a következőképpen történik tehát:

a0e03 = 10100000111000000011
10100000111000000011>>8 = 101000001110
0x1f = 0001 1111
101000001110 & 0001 1111 = 000000001110 (14 decimális)

A végeredmény így $0.5^{14} = 0.00006104$ J.

A time units kiszámításához ugyanezt a módszert használjuk, annyi, hogy 8 helyett 16 bittel fogjuk eltolni az értéket, valamint 0x1f helyett újra 0xf-el számolunk hiszen csak 4 bitet használunk. A végeredmény egyébként 0.97656ms lesz, így tehát a sok helyen dokumentált 1ms frissítési idő sokkal inkább közelebb van ehhez a végeredményhez.

```
1. time_units=pow(0.5,(double)((result>>16)&0xf));
```

Ha ez megvan, akkor már kezelhetjük a teljesítménymérő regisztereket, először is kiolvassuk az eredményt (package esetében):

```
1. result=read_msr(fd,MSR_PKG_ENERGY_STATUS);  
2. package_before=(double)result*cpu_energy_units;
```

Ezek után következhet a mérendő kódrészlet, majd újból kiolvassuk az eredményt:

```
1. result=read_msr(fd,MSR_PKG_ENERGY_STATUS);  
2. package_after=(double)result*cpu_energy_units;  
3. printf("\t\tPackage energy: %.6fJ\n",  
4.    package_after-package_before);
```

A háttérben mind a három módszer esetén így kapjuk meg az eredményeket, és ezért nincs különösebb lényege, hogy melyiket választjuk.

A kiválasztott modell (esetünkben perf event) alapján már implementálhatjuk a programunkat. Segítségünkre lesz egy C-ben írt modul.^[15]

¹⁵ web.eece.maine.edu/~vweaver/projects/rapl/rapl-read.c

Láthatjuk a 2.2-es pontban említett három módszer függvényeit is:

```
1. static int rapl_msr(int core, int cpu_model);
2. static int rapl_perf(int core);
3. static int rapl_sysfs(int core);
```

Mivel a szakdolgozat során a perf event interfészt használjuk, és lényegi különbség a funkcionalitásukban nincsen, valamint az eredmények is azonosak mindhárom funkció esetén, ezért a továbbiakban a *rapl_perf* függvényt fogjuk tüzetesebben vizsgálni.

Lényegében három pontra szedhetjük szét a mérést:

1. syscall() segítségével meghívjuk a perf parancsot, elindítva így a mérést

Ez a kódunkban a következő képpen néz ki:

```
1. fd[i][j] = perf_event_open(&attr, -1, package_map[j], -1, 0);
```

Ha Visual Studio Code IDE-t használunk, akkor F12 gombot nyomva a *perf_event_open* néven egyből eljutunk a függvény definíciójához:

```
1. static int perf_event_open(struct perf_event_attr *hw_event_up,
    pid_t pid, int cpu, int group_fd, unsigned long flags) {
2.
3.     return syscall(__NR_perf_event_open, hw_event_uptr, pid,
        cpu, group_fd, flags);
4. }
```

A syscall egy szolgáltatáskérés, amelyet egy program intéz a kernelhez. Legjobb példa erre a *printf()* függvény ami lényegében a *write()* hívás wrappere. Itt ugyanez történik, csak nem a write függvényt hívjuk meg, hanem a *perf_event_opent*, aminek a paraméterezése a következő linken érhető el: [perf event open\(2\)- Linux manual page](#).

Ez a függvény a programunkban egy file descriptorral fog visszatérni, amit az `fd` változóban fogunk tárolni.

2. mérendő kódrészlet

Az alap programban a mérendő kódrészletnél csak egy sima `sleep` műveletet mérünk:

```
1. printf("\n\tSleeping 1 second\n\n");  
2. sleep(1);
```

Természetesen itt bármit mérhetünk, tesztelhetünk, nem igazán van megkötve a kezünk.

A konténerek, függvények mérését is itt fogjuk elvégezni.

3. leállítjuk a mérést és kiolvassuk az eredményt

Ezt a következőképpen érhetjük el:

```
1. read(fd[i][j], &value, 8);  
2. close(fd[i][j]);
```

Itt a `perf_event_open` függvénnyel létrehozott file descriptorok segítségével ki tudjuk olvasni a végeredményt, majd bezárjuk a fájlt.

Ezek után akár már ki is írhatjuk az eredményt a standard outputra:

```
1. printf("\t\t%s Energy Consumed: %lf %s\n",  
2.     rapl_domain_names[i],  
3.     (double)value*scale[i]);
```

A `rapl_domain_names` egy globális változó:

```

1. char rapl_domain_names[NUM_RAPL_DOMAINS][30]= {
2.     "energy-cores",
3.     "energy-gpu",
4.     "energy-pkg",
5.     "energy-ram",
6.     "energy-psys",
7. };

```

A *value* egy lokális változó, ami lényegében a *perf_event_open* függvény által létrehozott fájlból kiolvasott értéket veszi fel.

```

1. long long value;

```

Természetesen a fenti kódok mind egy vagy két for ciklusban zajlanak, innen is jönnek az *i* illetve *j* változók, de a lényeg ezekben a sorokban vannak.

3.3 Tesztelés

A tesztelés során két fő koncepciót kellett szem előtt tartani. Az első, a mérések validálása, a második pedig a mérések pontossága.

Validálás

A validálás során arra a kérdésre kerestük a választ, hogy a programunk valóban jól működik-e, illetve, hogy valóban azt mérjük, amit szeretnénk.

A tesztelés során először is ki kellett választani a megfelelő módszert a mérésre. Esetünkben ez a perf event interfész lett.

A különböző eventeket a következőképpen lehet lekérni:

```

1. $ perf list

```


Tehát például egy adott program fogyasztását a következőképpen tudjuk mérni terminálban:

```
1. $ sudo perf stat -a -e "power/energy-cores/" ./program
```

Ez a parancs leméri a Cores (magok) fogyasztását (*energy-pkg* = package mérése, *energy-gpu* = gpu mérése, *energy-ram* = ram mérése), valamint azt is leméri hogy milyen gyorsan futott le a program. Ezek után már csak arra van szükségünk, hogy ezt a perf parancsot valahogy egy C/C++ programból is megtudjuk hívni. Mivel a fenti parancsnak nincsen wrappere, ezért egy `syscall()` függvénnyel tudjuk csak ezt elérni.

Az így kapott eredményeket többféleképpen is validálhatjuk. Például megnézhetjük, hogyha csak egy sima sleep műveletet mérünk, milyen eredményt kapunk. Ezt érdemes megnézni több értékre is. Ha irracionálisan nagy eredményeket kapunk, akkor sejthető, hogy valami nincs rendben. Aztán összehasonlíthatjuk például ezt a sima sleep műveletet egy sokkal processzor igényesebb kóddal. Ha látható eltérés van az eredmények között akkor már jófelé haladunk. (az eltérés mértéke természetesen az adott kódtól függ.)

Mérések pontossága

A validálás után következhet a mérések pontossága.

Az egyik módszer az lehetne, hogy több különböző paraméterezéssel felszerelt rendszeren is megmérnénk a fogyasztásokat, viszont esetemben sajnos csak egy gép állt a rendelkezésemre, így ezt a módszert a szakdolgozat alatt nem tudtuk kihasználni, habár meg kell említenünk hogy több kutatás is foglalkozott már ezzel a témával.^[16]

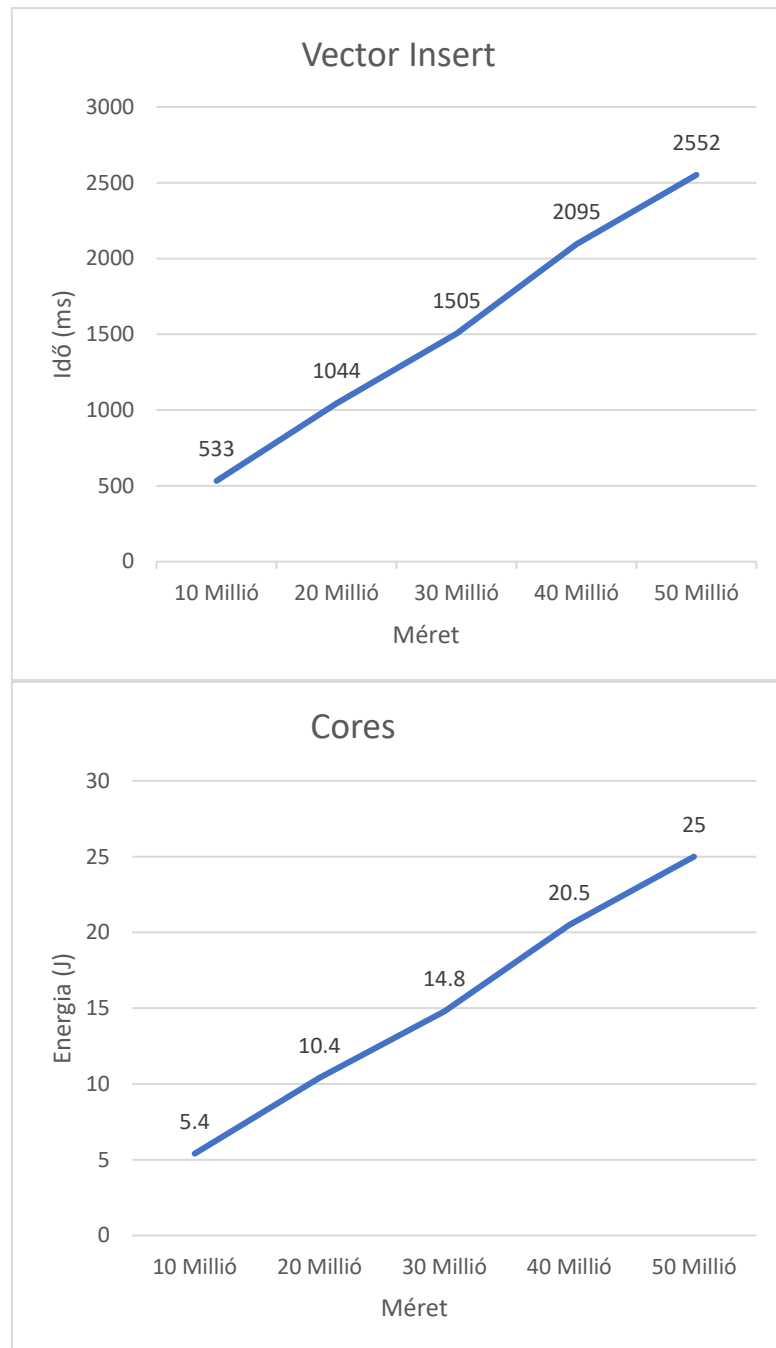
A másik módszer a rendszer szintű mérések pontosságának a figyelése, magyarul, ha adott rendszeren futtatjuk ugyanazt a kódot, a mérések eredményeinek, egy kis szórást eltekintve,

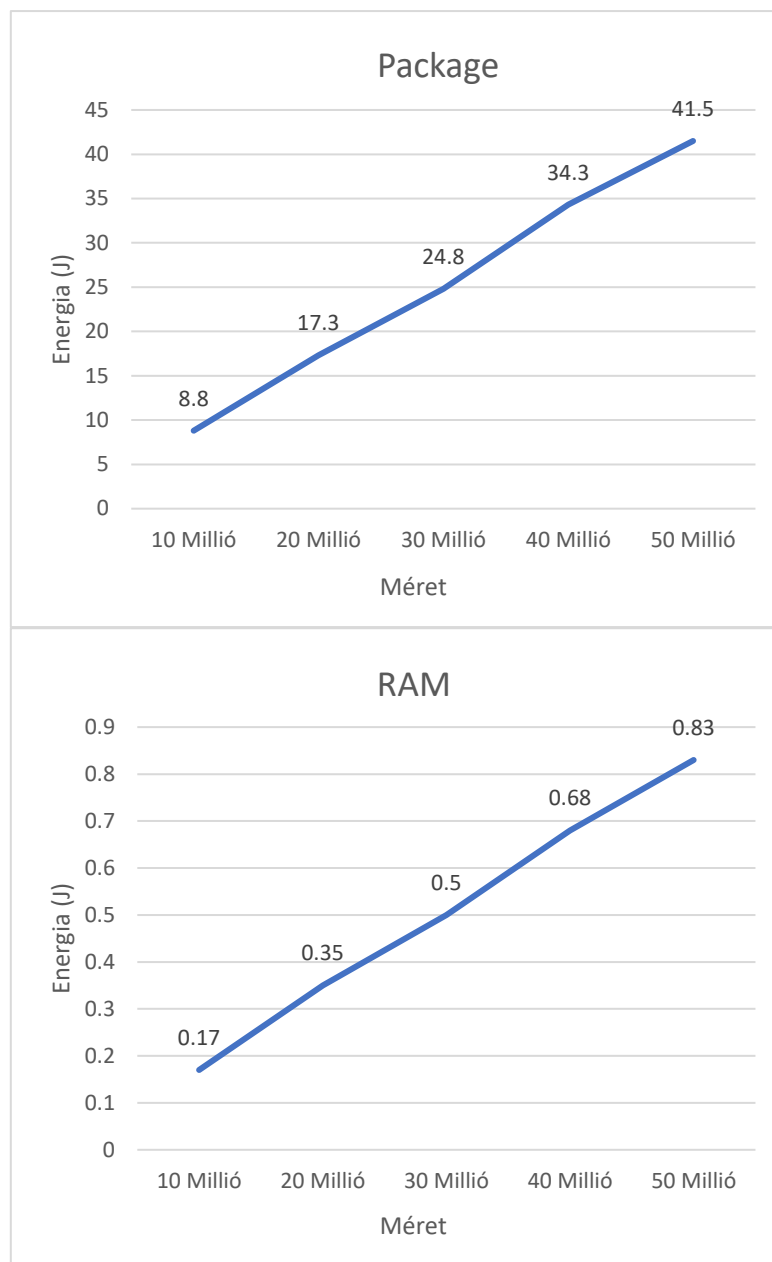
¹⁶ [*RAPL in Action: Experiences in Using RAPL for Power Measurements: ACM Transactions on Modeling and Performance Evaluation of Computing Systems: Vol 3, No 2*](#)

nem nagyon szabadna eltérniük. Azt már előre sejthetjük, hogy a legnagyobb eltérések a Package mérésénél lesznek, hiszen itt mérjük a legtöbb komponenst. Ez után következnek a magok, majd a RAM. Ha a szórásunk olyan 1-5% között mozog, akkor a méréseink pontosnak nevezhetőek. Fontos, hogy a szórást százalékban adjuk meg és ne egy pontos értéket, hiszen míg 0.5J szórás a Package esetében még bőven belefér az előbb említett határok közé, ez a RAM esetében már nem igaz, itt már óriási pontatlanságot mutatna. (persze a mérés nagyságától függ ez is)

A tesztelés során az implementált kód eredményeit is fogjuk vizsgálni.

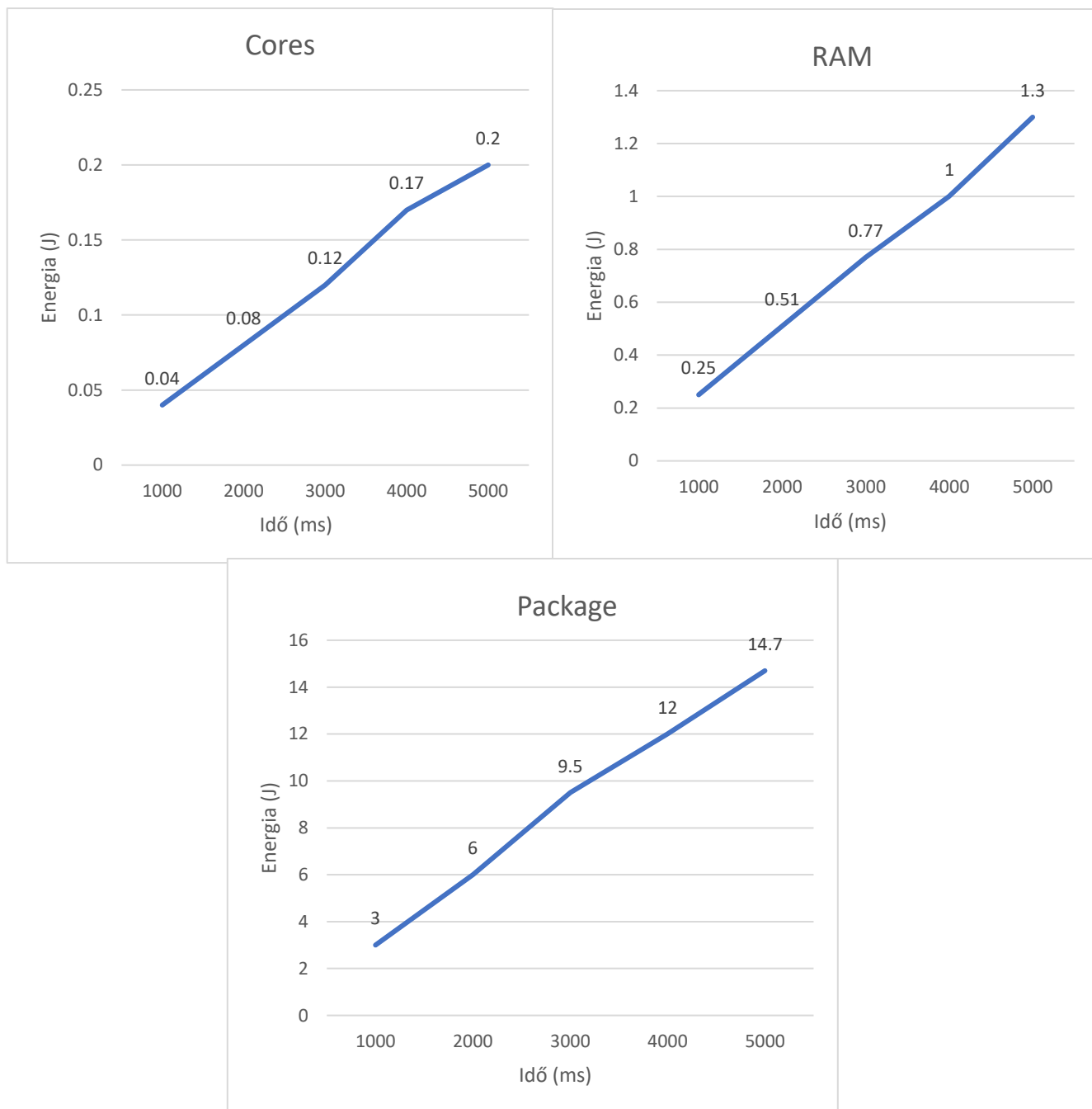
Az egyik mérés, amit mindenképpen meg kell nézni, az az, hogy különböző konténer méretre ugyanannak a metódusnak a fogyasztási eredményei hogyan reagálnak. Optimális esetben az eredményeknek egy egyenesre rá kéne férniük. A mérési eredmények a következő grafikonon láthatóak:





3.1 grafikon. Mérések eredményei

Jól látható hogy az eredmények ráférnek (kisebb eltéréssel) egy egyenesre.
Aztán megmérhetjük hogy “üresjáratban” (sleep) is megmarad-e ez a tulajdonság.



3.2 grafikon. Mérések eredményei

Láthatjuk, hogy a tulajdonság itt is megmarad.

Így akár már neki is állhatunk a konténerek, függvények méréséhez. Az alábbi táblázatok a mérések eredményeit tartalmazzák 10 milliós populációval nézve.

	vector		Alapfogyasztás*	Teljes Fogyasztás	deque		Alapfogyasztás	Teljes Fogyasztás
	J	ms	J	J	J	ms	J	J
insert (end)	Cores 5.565	533	Cores 0.035	Cores 5.6	Cores 3.97	398	Cores 0.03	Cores 4.0
	Pkg 5.6		Pkg 1.6	Pkg 7.2	Pkg 4.0		Pkg 1.2	Pkg 5.2
	RAM 0.04		RAM 0.14	RAM 0.18	RAM 0.01		RAM 0.10	RAM 0.11
push_back	Cores 2.085	204	Cores 0.015	Cores 2.1	Cores 1.585	154	Cores 0.015	Cores 1.6
	Pkg 2.09		Pkg 0.61	Pkg 2.7	Pkg 1.54		Pkg 0.46	Pkg 2.0
	RAM 0.05		RAM 0.05	RAM 0.1	RAM 0.01		RAM 0.04	RAM 0.05
push_front	N/A	N/A	N/A	N/A	Cores 1.358	128	Cores 0.012	Cores 1.37
	N/A		N/A	N/A	Pkg 1.37		Pkg 0.39	Pkg 1.76
	N/A		N/A	N/A	RAM 0.03		RAM 0.03	RAM 0.06
emplace_back	Cores 2.18	207	Cores 0.016	Cores 2.2	Cores 1.687	167	Cores 0.013	Cores 1.7
	Pkg 2.18		Pkg 0.62	Pkg 2.8	Pkg 1.7		Pkg 0.5	Pkg 2.2
	RAM 0.046		RAM 0.054	RAM 0.1	RAM 0.07		RAM 0.043	RAM 0.05
emplace_front	N/A	N/A	N/A	N/A	Cores 1.464	140	Cores 0.016	Cores 1.48
	N/A		N/A	N/A	Pkg 1.47		Pkg 0.43	Pkg 1.9
	N/A		N/A	N/A	RAM 0.022		RAM 0.038	RAM 0.06

3.1 táblázat. Mérések eredményei

*Az alapfogyasztást a tőle balra található ms érték alapján van számolva. Például: vector insert művelet esetén az alapfogyasztás 533ms alatt lett lemérve egy sima sleep művelettel.

	list		Alapfogya sztás	Teljes Fogyaszt ás	set		Alapfogya sztás	Teljes Fogyasztás
	J	ms	J	J	J	ms	J	J
insert (end)	Cores 7.855	744	Cores 0.045	Cores 7.9	Cores 18.07	1818	Cores 0.13	Cores 18.2
	Pkg 8.0		Pkg 2.2	Pkg 10.2	Pkg 18.5		Pkg 5.5	Pkg 24.0
	RAM 0.13		RAM 0.19	RAM 0.32	RAM 0.20		RAM 0.47	RAM 0.67
push_back	Cores 7.35	722	Cores 0.05	Cores 7.4				
	Pkg 7.33		Pkg 2.17	Pkg 9.5				
	RAM 0.12		RAM 0.19	RAM 0.31				
push_front	Cores 7.35	700	Cores 0.05	Cores 7.4				
	Pkg 7.4		Pkg 2.1	Pkg 9.5				
	RAM 0.12		RAM 0.18	RAM 0.3				
emplace_front	Cores 7.45	716	Cores 0.05	Cores 7.5				
	Pkg 7.45		Pkg 2.15	Pkg 9.6				
	RAM 0.11		RAM 0.19	RAM 0.3				
emplace_back	Cores 7.35	720	Cores 0.05	Cores 7.4				
	Pkg 7.44		Pkg 2.16	Pkg 9.6				
	RAM 0.11		RAM 0.19	RAM 0.3				

3.2 táblázat. Mérések eredményei

	Multiset		Alapfogyasztás	Teljes Fogyasztás	unorderedset		Alapfogyasztás	Teljes Fogyasztás
	J	ms	J	J	J	ms	J	J
insert (end)	Cores 17.38	1807	Cores 0.12	Cores 17.5	Cores 27.03	2730	Cores 0.17	Cores 27.2
	Pkg 17.8		Pkg 5.4	Pkg 23.2	Pkg 27.8		Pkg 8.2	Pkg 36.0
	RAM 0.22		RAM 0.46	RAM 0.68	RAM 0.3		RAM 0.7	RAM 1.0

3.3 táblázat. Mérések eredményei

Pontosság

A legfontosabb kérdés talán az lehet, hogy vajon mennyire pontosak ezek mérések? Nem meglepő módon a RAM fogyasztás mérése bizonyult a legpontosabbnak. Lényegében bármikor, bármilyen időközönként mérhettem, az eredmények szórása mindig 1-2% között mozgott. A Cores esetében ez az érték már inkább 1-5% közé esett, esetekben egy-egy kirívó érték, de ez nagyon ritka volt. A package esetében már más a helyzet. Itt már nem volt ritka, hogy az 5%-os szórást is átlépték az értékek. Azonban a jó hír az, hogy ilyenkor az alapfogyasztás is arányosan megnőtt. Magyarul a végeredmény (sötétebb kerettel rendelkező cellák) nem változott annyival, hogy átlépje az 5% határt. Erre majd visszatérünk a későbbiekben egy konkrét példával is.

Az alábbi táblázat pedig másodpercre lebontva adja meg a fogyasztást.

	vector	deque	list	set	multiset	unorderedset
insert (end)	Cores 10.44 J/s	Cores 9.97 J/s	Cores 10.56 J/s	Cores 9.94 J/s	Cores 9.65 J/s	Cores 9.9 J/s
	Pkg 10.5 J/s	Pkg 10.05 J/s	Pkg 10.75 J/s	Pkg 10.18 J/s	Pkg 9.85 J/s	Pkg 10.18 J/s
	RAM 0.075 J/s	RAM 0.025 J/s	RAM 0.17 J/s	RAM 0.11 J/s	RAM 0.12 J/s	RAM 0.11 J/s
push_back	Cores 10.22 J/s	Cores 10.3 J/s	Cores 10.18 J/s			
	Pkg 10.25 J/s	Pkg 10.0 J/s	Pkg 10.15 J/s			
	RAM 0.25 J/s	RAM 0.065 J/s	RAM 0.17 J/s			
push_front	N/A	Cores 10.61 J/s	Cores 10.5 J/s			
	N/A	Pkg 10.7 J/s	Pkg 10.57 J/s			
	N/A	RAM 0.23 J/s	RAM 0.17 J/s			
emplace_front	N/A	Cores 10.46 J/s	Cores 10.4 J/s			
	N/A	Pkg 10.5 J/s	Pkg 10.4 J/s			
	N/A	RAM 0.16 J/s	RAM 0.15 J/s			
emplace_back	Cores 10.53 J/s	Cores 10.1 J/s	Cores 10.21 J/s			
	Pkg 10.53 J/s	Pkg 10.2 J/s	Pkg 10.3 J/s			
	RAM 0.22 J/s	RAM 0.42 J/s	RAM 0.15 J/s			

3.4 táblázat. Másodpercre lebontott fogyasztás

Jól látható, hogy a package illetve Cores fogyasztás olyan 10 J/s körül mozognak. Viszont ne felejtsük el, hogy például a push_back esetében a deque ugyanannyi idő alatt majdnem hétszer annyi elemet tud feldolgozni. Ez az állítás pedig még a 3.3 szekció elején található grafikonok miatt igaz, ahol bebizonyítottuk, hogy az eltelt idő egyenesen arányos a container méretével.

Itt pedig a már előbb említett példa. Jól látható, hogy bár a második mérés esetében a package fogyasztása majdnem 20% növekedést ért el, mindez csak annak volt köszönhető, hogy az alapfogyasztás is ennyivel megnőtt. Ebből adódóan a végeredményünkben lényeges változás nem mutatkozott.

multiset								
Első mérés			Alapfogyasztás	Teljes Fogyasztás	Második mérés		Alapfogyasztás	Teljes Fogyasztás
	J	ms	J	J	J	ms	J	J
insert (end)	Cores 17.42	1807	Cores 0.08	Cores 17.5	Cores 17.5	1796	Cores 0.07	Cores 17.57
	Pkg 17.8		Pkg 5.4	Pkg 23.2	Pkg 17.6		Pkg 11.9	Pkg 29.5
	RAM 0.22		RAM 0.46	RAM 0.68	RAM 0.22		RAM 0.46	RAM 0.68

3.5 táblázat. Két külön mérés eredményei

multiset
<p>Cores</p> <p>9.65 J/s-9.74 J/s = 0.09 J/s = ~1% változás</p>
<p>Pkg</p> <p>9.85 J/s – 9.8 J/s = 0.05 J/s = ~1% változás</p>
<p>RAM</p> <p>0.12 J/s-0.12 J/s = 0.00 J/s = 0% változás</p>

3.6 táblázat. Másodpercre lebontott változás

Látható tehát, hogy a másodpercenkénti fogyasztásban lényeges különbség nincs.

Az ajánlásokat így tehát a kékekkel színezett táblázatok alapján érdemes megtenni.

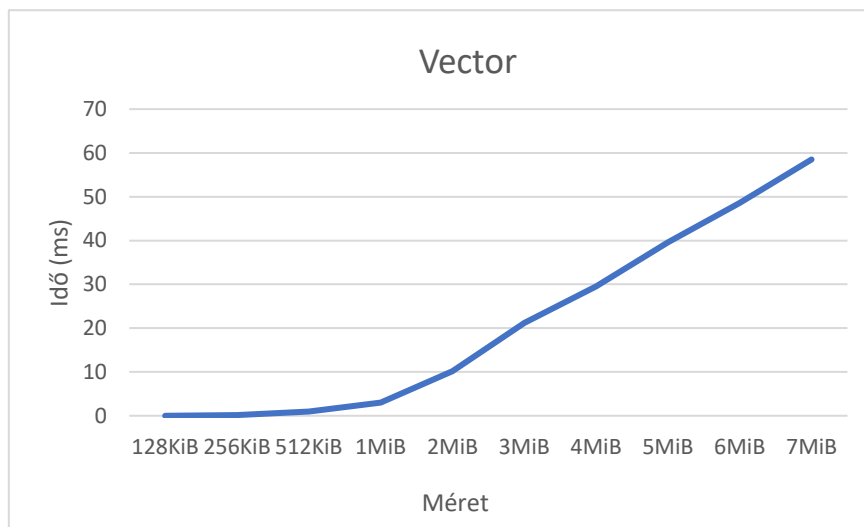
Ezek alapján a *deque* konténer szinte minden egyes függvényénél a leggyorsabb volt és így a legkevesebb energiát is fogyasztotta. Ezek után következik a *vector*, *list*, *set*, *multiset*, majd *unorderedset* konténerek.

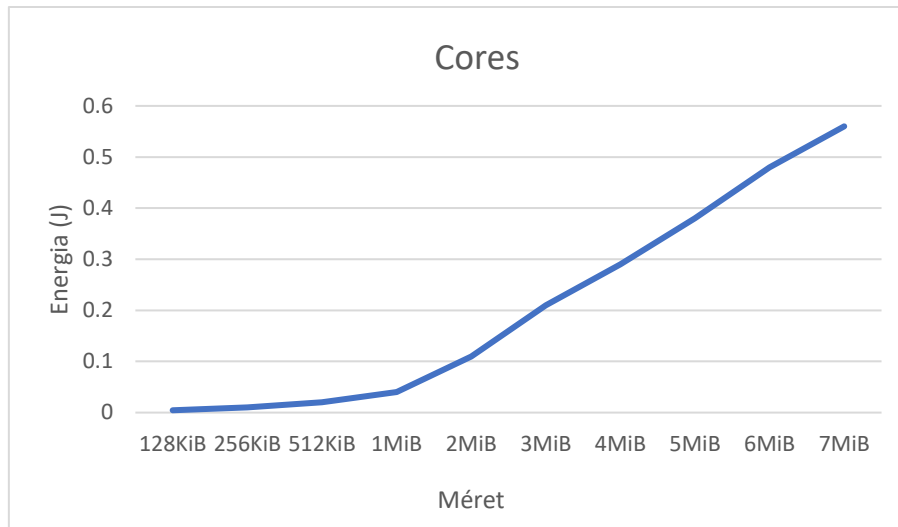
Cache hatás

A tervezési fejezetben megemlítettük, hogy érdemes lenne megnézni, hogy a RAPL képes-e észrevenni a cache hatásait a teljesítményen.

A tesztelés lényegében annyiból áll, hogy különböző méretű vektorokon végig megyünk, és minden elemét módosítjuk.

Az alábbi grafikonok megmutatják a mérési eredményeket:





3.3 grafikon. Cache hatásai

Jól látható, hogy kis méretű programoknál egyáltalán nem mindegy, hogy az illető 1MiB vagy 2MiB mérettel dolgozik. Azt is észrevehetjük, hogy 2MiB felett már egy egyenesre ráférnek az értékek, ezeket persze már a 3.1, illetve 3.2 grafikonon is láttuk.

4. Összegzés

A számítástechnika energiahatékonyságának vizsgálata már régóta aktuális téma. A RAPL bevezetése rengeteg új lehetőséget adott, olyanokat, amelyek korábban nem voltak lehetségesek. Habár a RAPL közel sem tökéletes, a Sandybridge-ben való bevezetése óta már sokat javult. A szakdolgozat során megnéztük részletesen az Intel RAPL interfész lehetőségeit, pontosságát, valamint a modell-specifikus regiszterek használatát is megvizsgáltuk.

További fejlesztési lehetőségek

Természetesen rengeteg lehetőségünk van a továbbiakban. Implementálhatunk egy grafikus felületet, ami segítené a programozóknak gyorsan dönteni, hogy egy adott helyzetben mely konténereket érdemes használni. Persze ehhez bővíteni kell az adatbázist újabb és újabb mérési eredményekkel.

Illetve érdemes lehet más programozási nyelvekre is implementálni a méréseket.

Egy kis érdekesség

Egy akadémiai kutatócsoport érdekes tanulmányt osztott meg arról, hogy az Intel CPU-k hogyan szivárogtathatnak ki adatokat. Konkrétan egy PLATYPUS nevű támadási stratégiát dolgoztak ki, amelyet arra lehet használni, hogy információkat nyerjünk ki az Intel eszközökből. A PLATYPUS tulajdonképpen a Power Leakage Attacks (Teljesítményszivárgási támadások) rövidítése: Targeting Your Protected User Secrets" (A védett felhasználói titkok megcélzása). Ahogy a kifejezés már sejteti, a támadási stratégia tulajdonképpen az energiafogyasztási különbségeket használja ki titkos adatok, például kriptográfiai kulcsok ellopására. Röviden, a PLATYPUS támadás egy Intel CPU-kat célzó side-channel támadás. A különbség itt azonban az, hogy a PLATYPUS-támadás távolról, szoftveren keresztül is végrehajtható. Ehhez a támadáshoz nem szükséges fizikai hozzáférés a céleszközhöz. A támadó az Intel RAPL (Running Average Power Limit) interfészének kihasználásával nyomon követheti az energiafogyasztás változásait. A RAPL látszólag energiamérőként szolgál, amely

lehetővé teszi a felhasználó számára a DRAM és a CPU energiafogyasztásának szoftveres felügyeletét és kezelését. A Grazi Műszaki Egyetem, a Birminghami Egyetem és a CISPA Helmholtz Információbiztonsági Központ munkatársai egy közzétett kutatási tanulmányukban felfedték, hogyan lehet egy Platypus-támadással megállapítani, hogy a RAPL-interfészen keresztül jelentett értékek alapján milyen adatokat dolgoznak fel egy CPU-ban.^{[17][18]}

¹⁷ [PLATYPUS: With Great Power comes Great Leakage \(platypusattack.com\)](https://platypusattack.com)

¹⁸ [PLATYPUS Attack: Novel Power Side-channel Attack Targets Intel Devices \(latesthackingnews.com\)](https://latesthackingnews.com)

Irodalomjegyzék

- [1] [\(PDF\) RAPL in Action: Experiences in Using RAPL for Power Measurements \(researchgate.net\)](#), Utolsó elérési dátum: 2020-05-03
- [2] [DRAM RAPL Validation \(maine.edu\)](#), Utolsó elérési dátum: 2020-05-03
- [3] [Energy measurements in Linux | chih's blog](#), Utolsó elérési dátum: 2020-05-03
- [4] [Linux support for Power Measurement Interfaces \(maine.edu\)](#), Utolsó elérési dátum: 2020-05-08
- [5] [Intel® 64 and IA-32 Architectures Software Developer Manual: Vol 3](#), Utolsó elérési dátum: 2020-05-07
- [6] [Processor power dissipation- Wikipedia](#), Utolsó elérési dátum: 2020-05-08
- [7] [How C++'s vector works: the gritty details « Frogatto & Friends](#), Utolsó elérési dátum: 2020-05-08
- [8] [c++- What really is a deque in STL?- Stack Overflow](#), Utolsó elérési dátum: 2020-05-08
- [9] [Difference between Vector and List in C++ – thispointer.com](#), Utolsó elérési dátum: 2020-05-08
- [10] [Containers library- cppreference.com](#), Utolsó elérési dátum: 2020-05-08
- [11] [An Introduction to Red-Black Tree | by Gild Academy — https://www.gildacademy.in/ | Medium](#), Utolsó elérési dátum: 2020-05-08

- [12] www.akkit.org/info/C++STANDARD-ISOIEC14882-1998.pdf, Utolsó elérési dátum: 2020-05-08
- [13] [EECS 311: STL Containers \(northwestern.edu\)](http://EECS.311:STLContainers.northwestern.edu) , Utolsó elérési dátum: 2020-05-08
- [14] [MSR RAPL POWER UNIT identifier- Linux source code \(v5.8-rc1\)- Bootlin](#), Utolsó elérési dátum: 2020-05-01
- [15] web.eece.maine.edu/~vweaver/projects/rapl/rapl-read.c, Utolsó elérési dátum: 2020-05-02
- [16] [RAPL in Action: Experiences in Using RAPL for Power Measurements: ACM Transactions on Modeling and Performance Evaluation of Computing Systems: Vol 3, No 2](#), Utolsó elérési dátum: 2020-05-04
- [17] [PLATYPUS: With Great Power comes Great Leakage \(platypusattack.com\)](#) , Utolsó elérési dátum: 2020-05-03
- [18] [PLATYPUS Attack: Novel Power Side-channel Attack Targets Intel Devices \(latesthackingnews.com\)](#) , Utolsó elérési dátum: 2020-05-03

Ábrajegyzék

1.1 ábra. Példa egy külső teljesítménymérőre.....	6
1.2 ábra. Mérendő tartományok	7
1.3 ábra. A programunk futás közben automatikusan megmondja a processzor típusát, valamint, hogy mennyi package-el, illetve maggal rendelkezünk.	8
2.1 ábra. Konténerek menüpont	12
2.2 ábra. Függvények menüpont.....	13
2.3 ábra. Teljes fogyasztás mérése	13
2.4 ábra. Alapfogyasztás mérése	14
2.5 ábra. Szinkronizációs hiba.....	16
3.1 ábra. MSR_RAPL_POWER_UNIT regiszter	17
3.2 ábra. MSR_PKG_ENERGY_STATUS MSR	18
3.3 ábra. MSR_PPO_ENERGY_STATUS MSR	18
3.4 ábra. MSR_DRAM_ENERGY_STATUS MSR	19
3.5 ábra. Vektor implementáció	21
3.6 ábra. Deque implementáció	22
3.7 ábra. List implementáció	22
3.8 ábra. Piros-fekete fa implementáció.....	23
3.9 ábra. MSR_RAPL_POWER_UNIT definiálásának helye.....	24
3.10 ábra. Catfish program grafikus felülete	25
3.11 ábra. Rdmsr parancs futtatása.....	26
3.12 ábra. A programunk futtatás után kiírja az egységeket.....	28