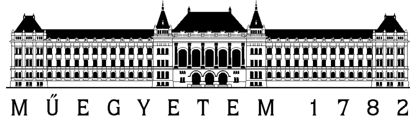


Teljesítményjellemzők vizsgálata

Szoftverfejlesztés laboratórium 2

Mérési segédlet

Készítette: Barta Patrik, Hajdu Ákos, Kocsis Imre, Vörös András

Verzió: 3.0

2020.

Budapesti Műszaki és Gazdaságtudományi Egyetem
Méréstechnika és Információs Rendszerek Tanszék

1 Bevezető

Az IT szolgáltatásokkal és megoldásokkal szemben támasztott követelmények közül a funkcionális megfelelőség után a gyakorlatban általában a „megfelelő” teljesítmény és rendelkezésre állás a legfontosabb. Durva általánosítással élve egy összetett szolgáltatás vagy egyedi alkalmazás teljesítményét a következő faktorok határozzák meg:

1. a feldolgozási/számítási logika,
2. a rendelkezésre álló erőforrások és
3. a munkaterhelés.

A tényleges „teljesítmény” különbségét az elvárttól mindhárom kategóriába tartozó hibaok okozhatja. Tekintsünk ezekre egy-egy példát.

1. Lehet egyszerűen a forráskód vagy a feldolgozást vezérlő konfiguráció „rossz”; egy rosszul skálázódó algoritmus, egy hibásan megvalósított szinkronizációs lépés, feleslegesen elvégzett tevékenységek vagy folyamat-virtuálisgépek esetén rosszul beállított halomméret mind okozhatnak teljesítményproblémákat.
2. A használni kívánt erőforrásokat parazita jelleggel foglalhatja más tevékenység; pl. egy asztali gép vírusirtójának ütemezett, teljes diszket érintő ellenőrzése minden I/O sávszélességre érzékeny alkalmazásra kihatással lehet.
3. Leginkább kiszolgálórendszerekben elképzelhető, hogy a hardver/szoftver konfiguráció nem alkalmas a jelentkező, előre nem látott méretű vagy jellegű terhelés megfelelő minőségű kiszolgálására.

A megfelelő teljesítmény biztosításához a célzott *teljesítménymérések* végzése és a megfigyelések kiértékelése alapvető fontosságú szinte minden esetben, hiszen szoftverrendszerek teljesítményét mérések nélkül, pusztán kódanalízissel vagy a kód/rendszerterv ismert teljesítményű példákkal való összehasonlításával felmérni általános esetben legfeljebb kvalitatívan tudjuk (vagy még úgy sem).

A teljesítmény megfigyelés alapú vizsgálata, mint igény általánosságából következik, hogy a teljesítményjellemzők vizsgálatára a gyakorlatban megközelítések és technológiák igen széles skáláját alkalmazzuk. A teljes spektrum bemutatása messze túlmutatna a foglalkozás keretein. A labor célja a hallgatók megismertetése a lokális teljesítménymérés és megfigyelés-kiértékelés két alapvető megközelítésével, a *profilinggal* és az *esemény-nyomkövetéssel* (*event tracing*). A labor végén továbbá a *párhuzamos programozás* témakörét is érintjük, amely gyakran alkalmazott módszer a teljesítmény javítására.

2 Profiling[1]

A profiling definiálható úgy, mint olyan dinamikus (tehát végrehajtáson és nem a forráskód vizsgálatán alapuló) programanalízis, melyet a végrehajtás (tesztelés) során gyűjtött információk alapján végzünk a program hibái és az optimalizálási lehetőségek felderítése érdekében. Két, a gyakorlatban nagy jelentőségű alkategóriája a *futásidő-* és a *memória-profiling*. Mindkét esetben a program futtatása során szeretnénk az analízis szempontjából potenciálisan fontos jellemzőket monitorozni (mint pl. metódusok hívásideje, hívási gyakorisága vagy elmaradt memóriafelszabadítások).

2.1 Futásidő profiling

Egy futó program profiling célú, szoftveres úton való megfigyelésére alapvetően két lehetőségünk kínálkozik; a *mintavételezés* és az *instrumentáció*. (A hardver alapú megközelítésekkel most nem foglalkozunk.)

[1] [2] felhasználásával.

2.1.1 Mintavételezés

Periodikusan *mintavételezhetjük* (*sampling*) egy futó program programszámlálóját, vagy ha arra van lehetőség, veremét; memória-profiling esetén a program által használt memóriaterületeket. Ezekből a pillanatképekből előállítható lesz a modulok és függvények hívási gyakorisága, a futtatásukkal töltött idő (vagy annak közelítése), a jellemző hívási láncok vagy a memóriaterületek evolúciója.

A mintavétel valamely értelemben periodikus kiváltásához használt események sokfélék lehetnek; legtöbbször adott időközönként vagy adott számú CPU ciklusonként mintavételezünk, de az órajelek helyett alkalmazhatjuk a laphibák, rendszerhívások vagy alacsony szintű processzor-események számát. (A pontos lehetőségeinket persze alapvetően a platform és a profiler képességei határozzák meg.)

Lényeges, hogy a mintavételezés alapú profiling nem egzakt eredményt ad, csak statisztikai közelítéseket. Így például egy ritkán meghívásra kerülő függvény a mintavételezés szerencsétlen időzítése miatt lehet, hogy nem is jelenik meg a regisztrált adatokban. Azt is látnunk kell, hogy mintavételezés esetén nem tudjuk, hogy egy függvény hányszor került meghívásra – amit regisztrálunk, az a megfigyelés, hogy a mintavétel pillanatában a végrehajtás az adott függvényben volt. Nem eldönthető, hogy két egymás után következő, azonos függvényre mutató minta felvétele között a végrehajtás a függvényben maradt-e vagy visszatért, és a függvény újra meghívásra került.

Másrészről azonban a mintavételezés alapú megközelítések jellemzően nem igénylik a kód / futtatókörnyezet felműszerezését, emellett hatásuk a rendszer futási teljesítményére alacsony és egyértelműen szabályozható.

2.1.2 Instrumentáció

A futtatott kód *instrumentációja* (*instrumentation*) során olyan extra utasítások kerülnek beszúrásra és végrehajtásra, melyek a végzett műveletekről képesek a profiler eszközt „tájékoztatni”. Ez koncepcionálisan igen hasonló ahhoz, mint amikor (a nagyon) ad-hoc hibakeresés során pl. egy metódusba / függvénybe belépés után és kilépés előtt a program valamely bemeneti/kimeneti folyamára a hibakeresést segítő „itt voltam” üzeneteket írat ki a fejlesztő[2]. Az instrumentáció, azaz az „extra” hívások beszúrása, a következő szinteken történhet [3]:

1. a forráskód rendelkezésre állása esetén közvetlenül a forráskódon vagy a linkelés/fordítás során;
2. forráskód hiányában
   a. menedzselt környezetekben a bytekód kontextusában, közvetlenül a bytekódon vagy futtatás közben az interpreter/virtuális gép képességeit kihasználva,
   b. nem menedzselt környezetekben statikusan a bináris kódon vagy a végrehajtás során különböző technikákat alkalmazva (pl. hardver debug támogatás, hívási táblák átírása).

Az egyszerűség kedvéért itt az instrumentáció kategóriája alá soroljuk a menedzselt környezetek (mint a Java Virtual Machine és .NET Common Language Runtime) azon képességét, hogy profilozó alkalmazások számára meghatározott API-n keresztül lehetővé teszik callback metódusok regisztrálását olyan virtuális gép szintű eseményekre, mint metódusba vagy szálba belépés, azokból kilépés vagy osztályok betöltése és kiürítése.

2.1.3 Megjelenítés és analízis

Akár mintavételezést, akár instrumentációt választunk, a kinyert megfigyeléseket a profilingot végző eszköz *regisztrálja*, majd jellemzően lehetőséget nyújt azok megjelenítésére is. Futásidő-profiling esetén például szokásosan legalább a *hívási fa* (*call tree*) és a *„kiterített” profil* (*flat profile* vagy *functions*) nézetek kerülnek előállításra és megjelenítésre.

[2] Ezen megoldások helyett a gyakorlatban természetesen naplózó, nyomkövető illetve profiling technikák alkalmazását javasoljuk inkább.

2.1.3.1 Hívási fa (call tree)

A hívási fa a metódusokban/függvényekben eltöltött idő top-down bontását adja meg. Az 1. ábra a labor során használt Visual Studio profiler egy példa-futására mutat egy hívási fa megjelenítést[3].

Function Name	Number of Calls	Elapsed Inclusive Time %	Elapsed Exclusive Time %	Avg Elapsed Inclusive Time	Avg Elapsed Exclusive Time
ConsoleFilterDemo.exe	0	100.00	0.00	0.00	0.00
ConsoleFilterDemo.Program.Main(string[])	1	99.12	0.02	2,352.87	0.52
AForge.Imaging.Filters.IFilter.Apply(class Syst	1	96.94	96.94	2,300.96	2,300.96
System.Drawing.Image.FromFile(string)	1	1.54	1.54	36.61	36.61
AForge.Imaging.Filters.OilPainting..ctor()	1	0.47	0.47	11.11	11.11
ConsoleFilterDemo.Program.someFunc2()	1	0.07	0.00	1.55	0.00
System.Console.WriteLine(string)	1	0.07	0.07	1.55	1.55
System.Console.WriteLine(string)	2	0.06	0.06	0.76	0.76
ConsoleFilterDemo.ConsoleFilterEventSource	1	0.01	0.01	0.35	0.34
ConsoleFilterDemo.Program.someFunc1()	1	0.00	0.00	0.11	0.00

1. ábra Hívási fa példa

Látható, hogy a teljes futásból generált hívási fa az idődimenziót kihagyja; minden hívási mélységben csak azt jeleníti meg, hogy az adott szint a további függvényeket hányszor hívta a futás során. A fa minden csomópontjánál az összes hívásra átlagolva és összegezve is kiszámításra kerül az ún. „inkluzív” és az „exkluzív” eltelt idő. Az „exkluzív” idő az adott függvényhívás végrehajtása során eltelt idő, nem számítva a függvény által hívott további függvények végrehajtásának az idejét; az „inkluzív” idő a fa adott csomópontja által definiált részfa exkluzív idejeinek az összege.

A következőket érdemes megfigyelnünk:

1. A hívások között látjuk egy konstruktor lefuttatását is (.ctor). (Pontosabban egy példányinicializáló konstruktorét; a típusinicializáló konstruktor – ami pl. egy osztály statikus tagjainak példányosításához szükséges – jelölése .cctor lenne.)
2. A hívások között viszont csak a „saját kódot” látjuk, a profiling a programban használt képfeldolgozó könyvtárba és a `System.Console.WriteLine`-t megvalósító `mscorlib.dll`-be nem „lépett át”; ez annak fényében, hogy explicit instrumentáción alapuló mérést kértünk, nem is meglepő. Mintavételezett esetben lehetőségünk lenne ezek belső futásidő-viszonyaiba is betekintést nyernünk[4].
3. Figyeljük meg, hogy a `System.Console.WriteLine` hívás két szinten is megjelenik (egyszer a `Main`-ből, egyszer pedig a `someFunc2`-ből)! Arra a kérdésre így a hívási fából (fa, és nem DAG volta miatt) nem mindenképp kapunk választ, hogy egy adott függvényben összesen mennyi ideig tartózkodott a program.

A hívási fa egy profiling futás során „legaktívabb” ágára az angol terminológia „*hot path*”-ként („forró út”) hivatkozik.

2.1.3.2 Functions

A Functions nézet (korábban flat profile) egy táblázat, mely a teljes programvégrehajtást függvénycentrikusan írja le; ugyanazokkal a metrikákkal, mint a hívási fa esetén, de azokat az egyes függvényekre származtatva (lásd 2. ábra).

Function Name	Number of Calls	Elapsed Inclusive Time %	Elapsed Exclusive Time %	Avg Elapsed Inclusive Time	Avg Elapsed Exclusive Time
AForge.Imaging.Filters.IFilter.Apply(class System.Drawing.Bitmap)	1	97.04	97.04	2,182.08	2,182.08
System.Drawing.Image.FromFile(string)	1	1.52	1.52	34.28	34.28
System.Diagnostics.Tracing.EventSource..ctor()	1	0.70	0.70	15.74	15.74
AForge.Imaging.Filters.OilPainting..ctor()	1	0.52	0.52	11.60	11.60
ConsoleFilterDemo.Program.Main(string[])	1	99.25	0.08	2,231.86	1.81

2. ábra "Flat profile" példa

[3] Instrumentált futás a „kicsi” függvények (esetünkben pl. a `someFunc1` és `someFunc2`) az instrumentációból való kihagyását kikapcsolva.

[4] Ehhez persze szükséges lehet jónéhány, a futtatott (és nem fejlesztés alatt álló) binárisokban már nem szereplő adat, legfőképp olyan *szimbólumok*, mint a függvények neve. Javasolt olvasmányok: [5], [6] és [4].

2.1.3.3 Analízis

A futásidő-profiling segítségével való teljesítmény-javításra általános recept nehezen lenne adható. Néhány szokásos hibaok, melyek jelenlétének feltárását a futásidő-profiling hatékonyan támogathatja, a következő:

1. **Haszontalan számítások.** Példa: nem törölt, de futtatott, a működés szempontjából már haszontalan „régi” kódrészletek.
2. **Felesleges újraszámítás.** Példa: részeredmények újraszámítása tárolás helyett.
3. **Rendszerszolgáltatások mértéktelen igénybevétele.** Példa: a rendszerhívások (*syscall*) processzor szintű kontextusváltásokat (*context switch*) okoznak (modern operációs rendszerekben ma már ez lehet részleges és szoftveresen megvalósított is). A szálak és folyamatok közötti, operációs rendszer ütemező által irányított váltásnak szintén kontextusváltás-költsége van. Így mind a túl gyakori rendszerhívások, mind a fizikai szálak számához képest aránytalanul nagyszámú és nem blokkolt folyamat/szál teljesítmény-problémákhoz vezethet, különösen valósidejű rendszerekben.
4. **Foglalt erőforrásokra várakozás.** Egy végrehajtási szál által egy függvényhívásban eltöltött idő magas lehet azért is, mert a végrehajtás szinkronizációra vagy pl. kommunikációs/állomány erőforráshoz való hozzáférésre vár.

Egy alkalmazás teljesítmény-problémáit az alkalmazás szemszögéből nézve külső források is okozhatják:

- A logikai és fizikai erőforrásoknak az operációs rendszerrel és az azon futó egyéb alkalmazásokkal, szolgáltatásokkal megosztott használata miatt a platform túlterhelése – pl. normál CPU vagy diszk-sávszélesség szaturáció, „megszakítás-vihar” (*interrupt storm*) által túlterhelt CPU – vezethet teljesítményproblémákhoz.
- Hasonlóan felmerülhetnek a futtató hoszton kívüli faktorok; például egy átviteli hibák miatt abnormálisan lecsökkent áteresztőképességű, az alkalmazás által használt hálózati kapcsolat.
- Az alkalmazás használhat olyan (távoli vagy helyi) szolgáltatásokat, melyek teljesítménye valamely okból kifolyólag nem megfelelő; ebben az esetben ezek teljesítményhibája átterjedhet az alkalmazásra.

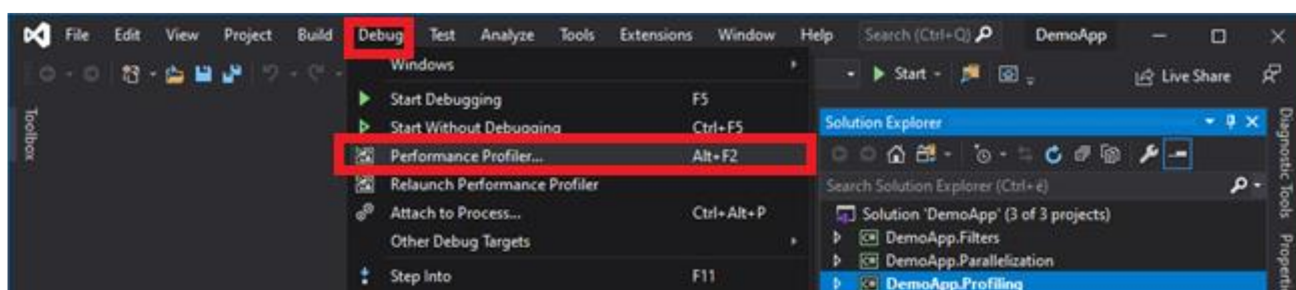
Profiling jellegű mérésekkel persze ezen esetekben is sokszor felderíthető az alkalmazás hibaállapotának jellege, és a valószínűsíthető hibaokok köre szűkíthető; „bedugult” diszksávszélesség esetén pl. észlelhetjük, hogy a szinkron állományíró és –olvasó hívások sokkal tovább tartanak, mint a (remélhetőleg létező) referenciakísérletek során.

2.1.4 .NET profiling a Visual Studio 2019 Community segítségével

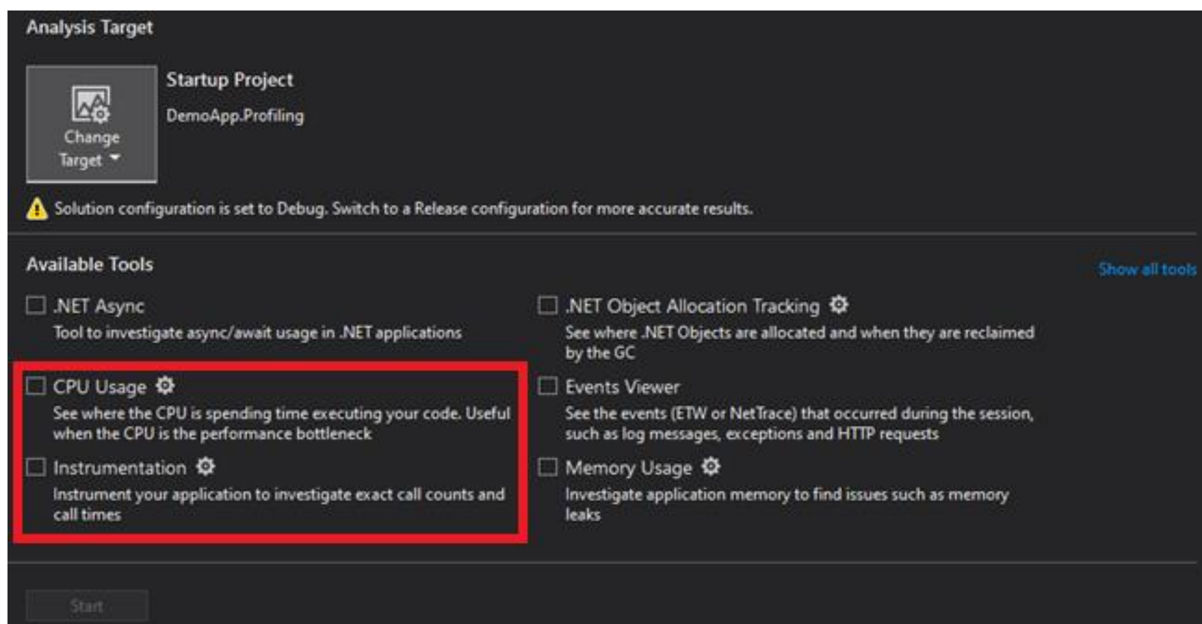
A segédlet és a mérés céljain túlmutat az elterjedt platformokon használt profiler-technológiák kimerítő tárgyalása és kategorizálása. Az olvasónak javasoljuk a Wikipedia „List of performance analysis tools” szócikke [15] áttekintését, mely jól érzékelteti az elérhető eszközök sokszínűségét.

A mérés egy reprezentatív technológia, a Microsoft Visual Studio 2019 Community segítségével ad bevezetést az (alkalmazás) profiling, mint általános megközelítés területére. A Visual Studio 2019 Community verzió beépített, az IDE-vel integrált profiling támogatással rendelkezik.

Egy megnyitott (Visual C#) projektre a `Debug` → `Performance Profiler` menüpontból érhetőek el a projekten (illetve az aktuális profiling munkameneten) értelmezett profiling akciók. Az „Instrumentation” értelemszerűen az instrumentációnak felel meg, a mintavételezés (sampling) pedig a „CPU Usage” kiválasztásával érhető el. A megfelelő módszer kiválasztása után egy felugró ablakban még a projektet is meg kell adni, amire a profiling futni fog.

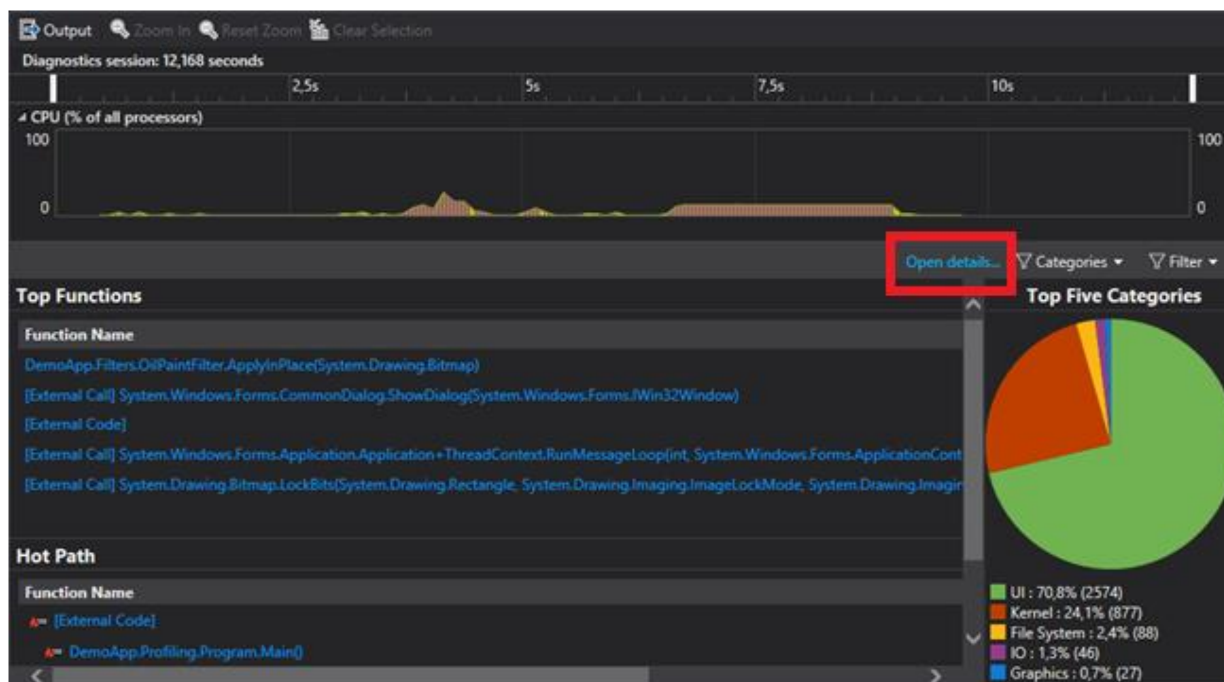


3. ábra Visual Studio: performance profiler indítása

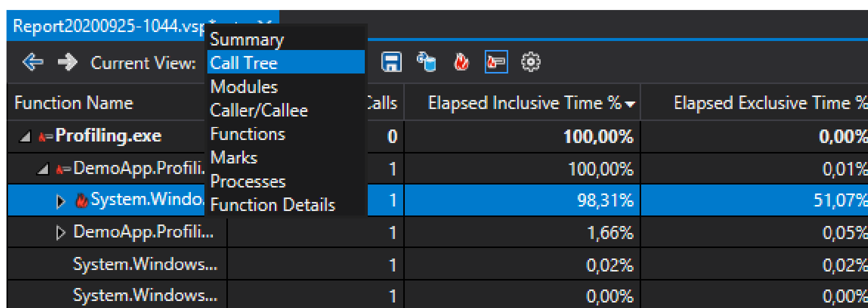


4. ábra Profiling akciók áttekintése

Az alkalmazás futtatása és bezárása után az elkészült teljesítményjelentéseket önálló lapokon nyithatjuk meg, melyeken belül különböző megjelenítési nézetek érhetőek el (6. ábra). Mintavételezés esetén a képernyő közepén található „Open details...” linken (5 ábra) érhetőek el a részletesebb nézetek.

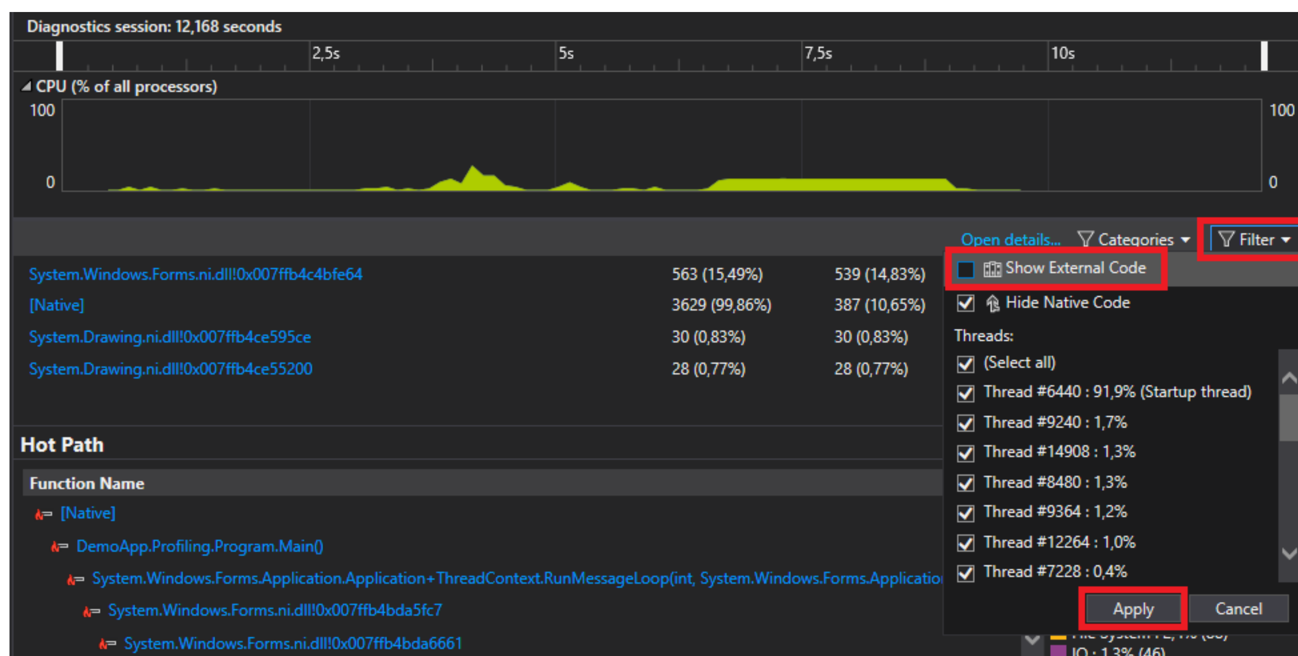


5. ábra Teljesítményjelentés áttekintő nézete mintavételezés esetén



6. ábra Teljesítményjelentés megjelenítési nézetei instrumentáció esetén

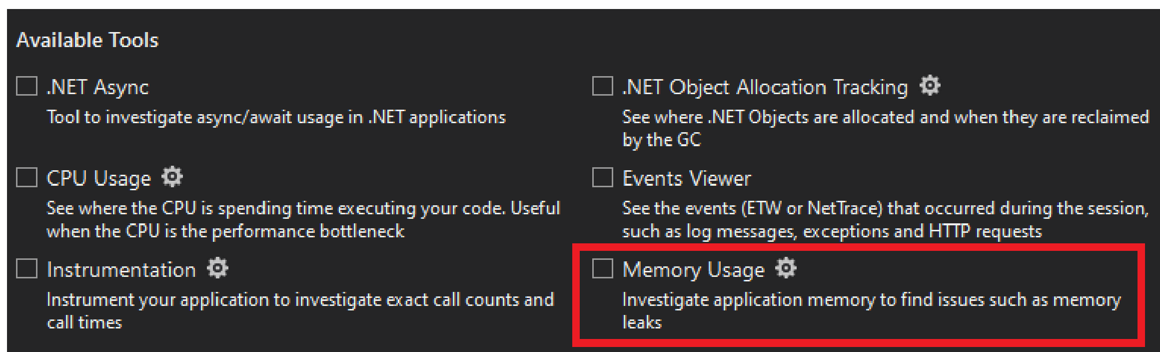
A legaktívabb futási ág (hot path) mintavételezés és instrumentáció esetén is már az áttekintő képernyőn elérhető, de a részletesebb hívási fa (call tree) nézetben is megtalálható. Mintavételezés esetén érdemes kipróbálni a külső kód megjelenítését is (Show External Code), amely az áttekintő nézetben a Filter menüpont alatt kapcsolható be (7. ábra).



7. ábra Külső kód megjelenítése mintavételezés esetén

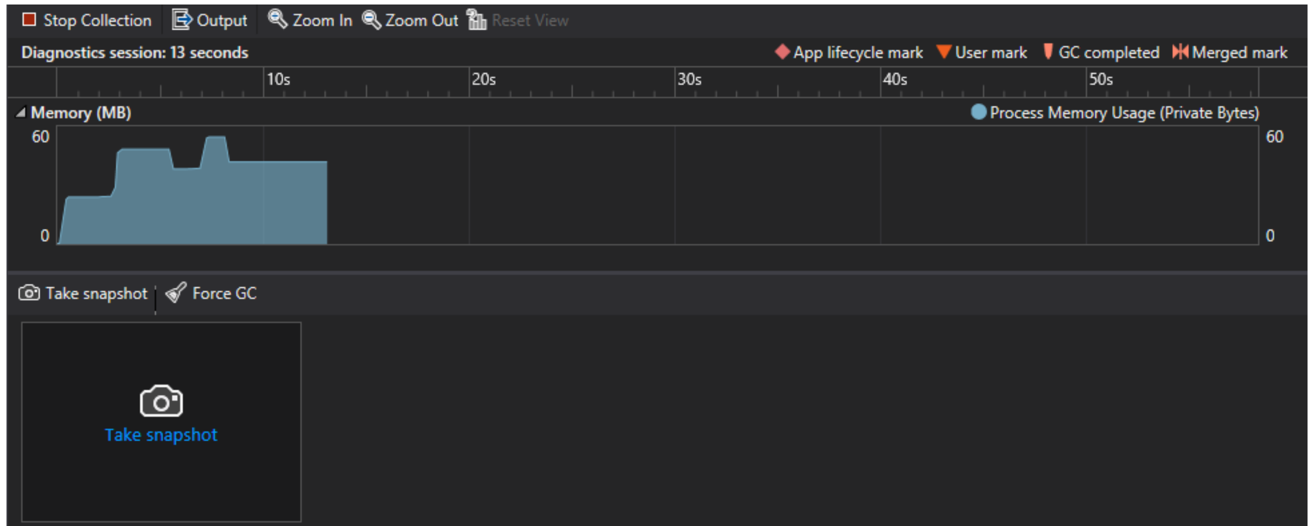
2.2 Memória profiling

A Visual Studio 2019 Community segítségével képesek vagyunk rögzíteni egy alkalmazás futása során a memóriafogyasztást, képesek vagyunk snapshot-okat készíteni és a Garbage Collector-t manuálisan elindítani. Ezeknek az eszközöknek a segítségével képesek vagyunk a memória problémák pontosabb azonosítására. Ez a lehetőség a mintavételezéshez és az instrumentációhoz hasonlóan a Visual Studio `Debug` → `Performance Profiler` menüpont alatt érhető el, „Memory Usage” néven (8. ábra).



8. ábra Memóriafogyasztás vizsgálatának indítása

A következő ábra mutatja a futás közben használt felület főbb elemeit.

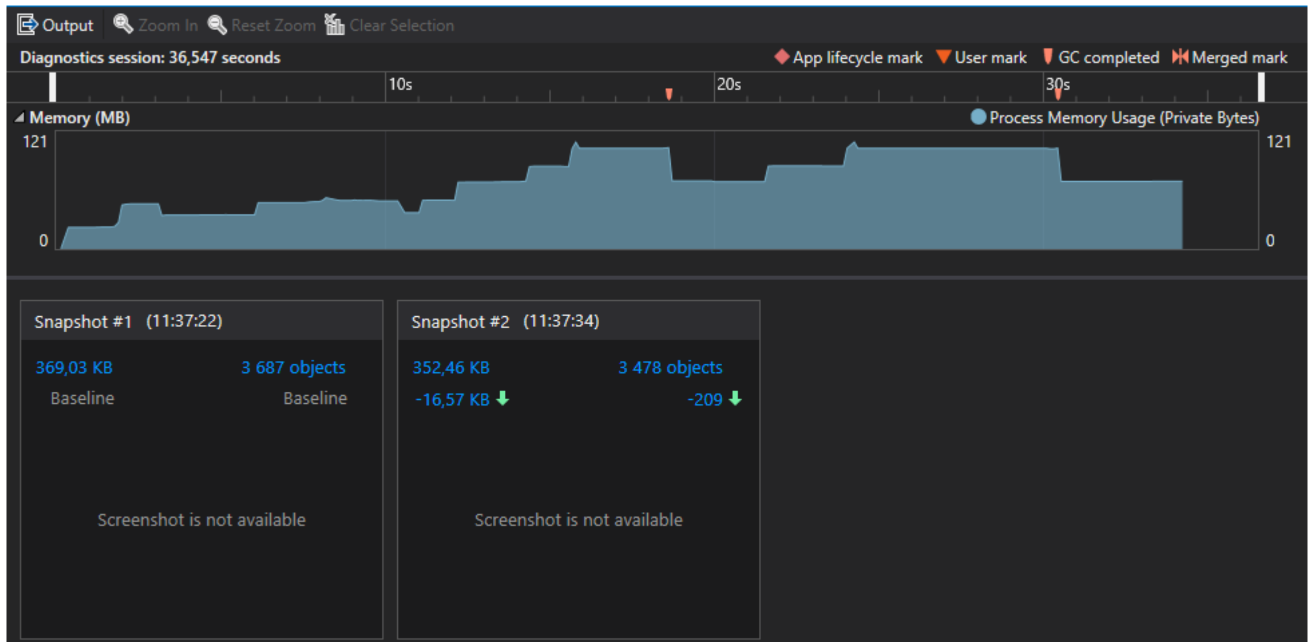


9. ábra Futás közbeni felület főbb elemei

A felső és középső menüsoron láthatóak a legfontosabb gombok:

- `Stop Collection`: Leáll a vizsgálat. (Ha kilépünk az alkalmazásból, akkor is leáll.)
- Take Snapshot: Ezzel rögzíthetjük a memória állapotát egy tetszőleges pillanatban.
- Force GC: Ezzel kényszeríthetjük ki a Garbage Collector futását.

A következő ábrán lehet látni a futás utáni eredmény ablakot.



10. ábra Futás utáni eredmény ablak

Amit érdemes észrevenni, hogy a felső idősávon látható, hogy mikor futott a GC. A diagram alatt a futás során rögzített snapshot-ok láthatóak, ezekre kattintva részletes adatokhoz juthatunk.

3 Esemény-nyomkövetés[5]

Az esemény-nyomkövetés definiálható úgy, mint olyan dinamikus program- és rendszeranalízis-technika, mely során nyomokat (*trace*) – események sorozatát – rögzítünk. Az események olyan logikai vagy fizikai aktivitásokat reprezentálnak, melyek az analízis szempontjából lényegesek lehetnek. A következő tulajdonságok tipikusan rögzítésre kerülnek az esemény-nyomkövetés során.

1. *Mi* történt (pl. esemény-azonosító segítségével).
2. *Mikor* történt az esemény (időbélyeg).
3. *Hol* történt az esemény (programfutás-események esetén pl. modul, függvény, kódsor).
4. Az esemény bekövetkeztének körülményeit meghatározó további adatok.

Az események időbélyegének rögzítése megtartja az események időbeli sorrendezését; az események helyével együtt így az esemény-nyomkövetés a végrehajtás-vezérlés és rendszerkomponensek interakciójának vizsgálatát is lehetővé teszi. Kiemelendő az is, hogy a profiling a tranziens jelenségeket elrejtheti – pl. egy igen sokszor futtatott, de néhány hívás során abnormálisan lassan visszatérő függvény úgy okozhat hibákat egy késleltetés-érzékeny rendszerben, hogy átlagos és összes megfigyelt futásideje a hibátlan viselkedést közelíti. Ezzel szemben a nyomkövetés lehetőséget ad az abnormális esetek felderítésére és szelektív vizsgálatára.

Az esemény-nyomkövetéssel potenciálisan megvalósítható analízisek magukban foglalják a profilingot is; pl. egy függvény-belépéseket és kilépéseket regisztráló nyomból a szokásos futásidő-profilok előállíthatóak.

Egyértelmű előnyei mellett az esemény-nyomkövetés rendelkezik néhány komoly hátránnyal a szigorúan vett profilinggal szemben.

1. **Implementációs többletköltség.** Az általában gyűjteni kívánt események nagy részéhez elkerülhetetlen az instrumentáció-fejlesztés. Jó példa erre azon alkalmazási szintű események köre, melyek a magas(abb) szintű alkalmazás-állapotot követik. Míg egy menedzselt platform esetén a függvénybelépés/kilépés instrumentációk akár dinamikusan és automatikusan injektálhatóak, az „inicializálás megkezdése/befejezése", „feladat megkezdése/befejezése" jellegű események instrumentációját a fejlesztőnek kell implementálnia (vagy magas szintű modellből generálnia). Kernel-eseményekhez – pl. laphibák (*page faults*) vagy folyamat-indítások – pedig természetszerűleg csak megfelelő kernel-támogatással férhetünk hozzá.
2. **Nagymennyiségű tárolandó adat.** Profiling esetén elégséges lehet csak a profilt adó futási jellemzőket követni és tárolni (pl. egy alkalmazás által hívott minden függvény összes/átlagos futásideje). Nyomkövetés során azonban az egyedi eseményeket és attribútumaikat is tároljuk. A szoftveresemények (ideértve a kernelt is) időskálája könnyen lehet milli- vagy mikroszekundum nagyságrendű; így a nyomkövetés másodpercenként több MB-nyi adatot is generálhat.
3. **Teljesítmény-többletköltség.** Az események puszta generálása és regisztrálása rendelkezhet a profilingénál nagyobb teljesítmény-overheaddel, különösen nagyszámú eseményforrás használatakor, vagy ha az események frekvenciája nagy. (A modern esemény-nyomkövető platformokon azonban ez már nem jellemző.)

Egy alkalmazás fejlesztése során eldöntendő kérdés, hogy az alkalmazás támogassa-e az alkalmazási szintű esemény-nyomkövetést, és ha igen, milyen részletességgel, hiszen az alkalmazás belső eseményeinek megfigyelhetővé tétele fejlesztési többletköltséggel jár. A kérdések ismerősek lehetnek más kontextusból – hasonló dilemmákkal áll szemben a fejlesztő akkor is, mikor egy alkalmazás naplózási mechanizmusairól kell dönteni.

A profilinghoz hasonlóan mind a segédlet, mind a labor egy reprezentatívnak tekinthető technológiára fókuszál az esemény-nyomkövetés esetén is. Esetünkben ez az `Event Tracing for Windows`.

[5] Részben [11] 3.3 alfejezete alapján.

3.1 Event Tracing for Windows[6]

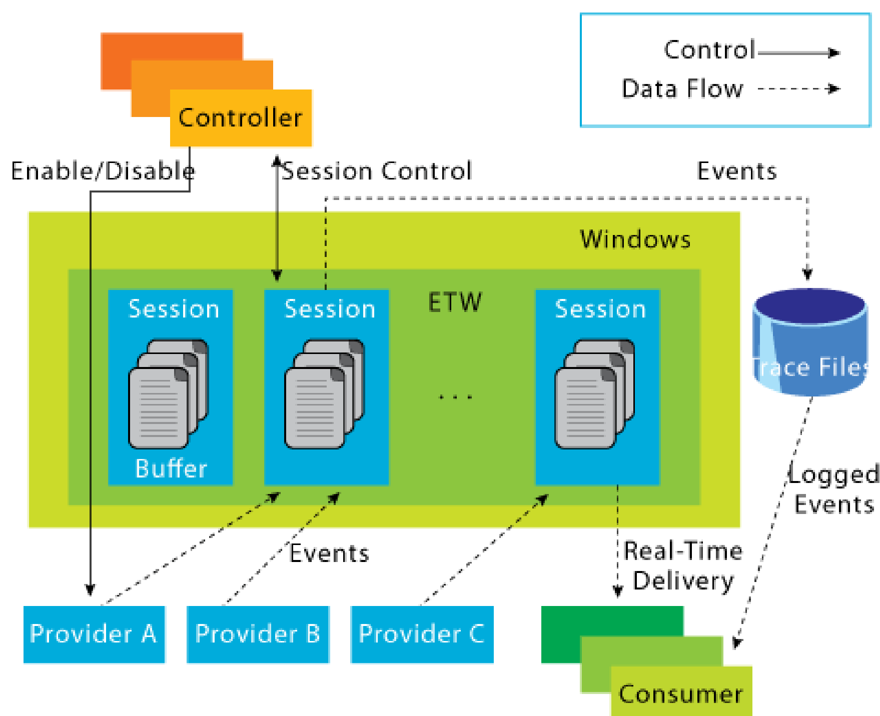
A mérés során esemény-nyomkövetési technológiaként a modern Microsoft Windows platformok `Event Tracing for Windows` (ETW) infrastruktúráját használjuk. Az ETW tervezése során az alapvető cél az volt, hogy általános nyomkövető platformot biztosítson felhasználói és kernel módú kód számára egyaránt úgy, hogy használata a nyomkövetett szoftver teljesítményére minimális hatással legyen.

Az ETW az eseményeket memória-bufferekbe írja, melyeket aszinkron módon ment diszkre (amennyiben azt egyáltalán igényeljük nyomkövetés közben). A diszkre írást dedikált kernel-szálak végzik, így azoknak legfeljebb az I/O sávszélesség-hatása befolyásolja a nyomkövetett rendszert, illetve alkalmazást.

Az ETW architektúra főbb komponensei a következők.

- **ETW munkamenetek (*ETW Sessions*).** A munkamenetek reprezentálják azokat a kernel-környezeteket, melyek a futó nyomkövetésekhez tartozó puffereket kezelik. Egy munkamenetben több szolgáltató eseményeit is gyűjthetjük, és párhuzamosan több munkamenet is futhat.
- **ETW szolgáltatók (*ETW Providers*).** Azon absztrakt, felhasználói vagy kernel módú komponensek, melyek az eseményeket szolgáltatják. A szolgáltatók általában esemény-kategóriához kötöttek (pl. `Microsoft-Windows-DHCPv6-Client`); egy szolgáltató eseményei származhatnak több mint egy végrehajtható állományból, dll-ből vagy meghajtómodulból. Egy szolgáltató több munkamenetbe is naplózhat.
- **ETW fogyasztók (*ETW Consumers*).** Az ETW által generált nyomokat feldolgozó és megjelenítő eszközök. Ilyen eszköz az `xperf` a `Windows 7` SDK-ban vagy a `Windows Performance Analyzer (WPA)` a `Windows 8 ADK`-ban.
- **ETW vezérlők (*ETW Controllers*).** A munkameneteket elindító és a szolgáltatóval összekapcsoló eszközök. Az `xperf` ETW vezérlőként is működik; szerepét a `Windows 8`-ban a `Windows Performance Recorder (WPR)` hivatott átvenni (bár az `xperf` egyelőre továbbra is használható opció).

A 11. ábra szemlélteti a komponensek kapcsolatait.



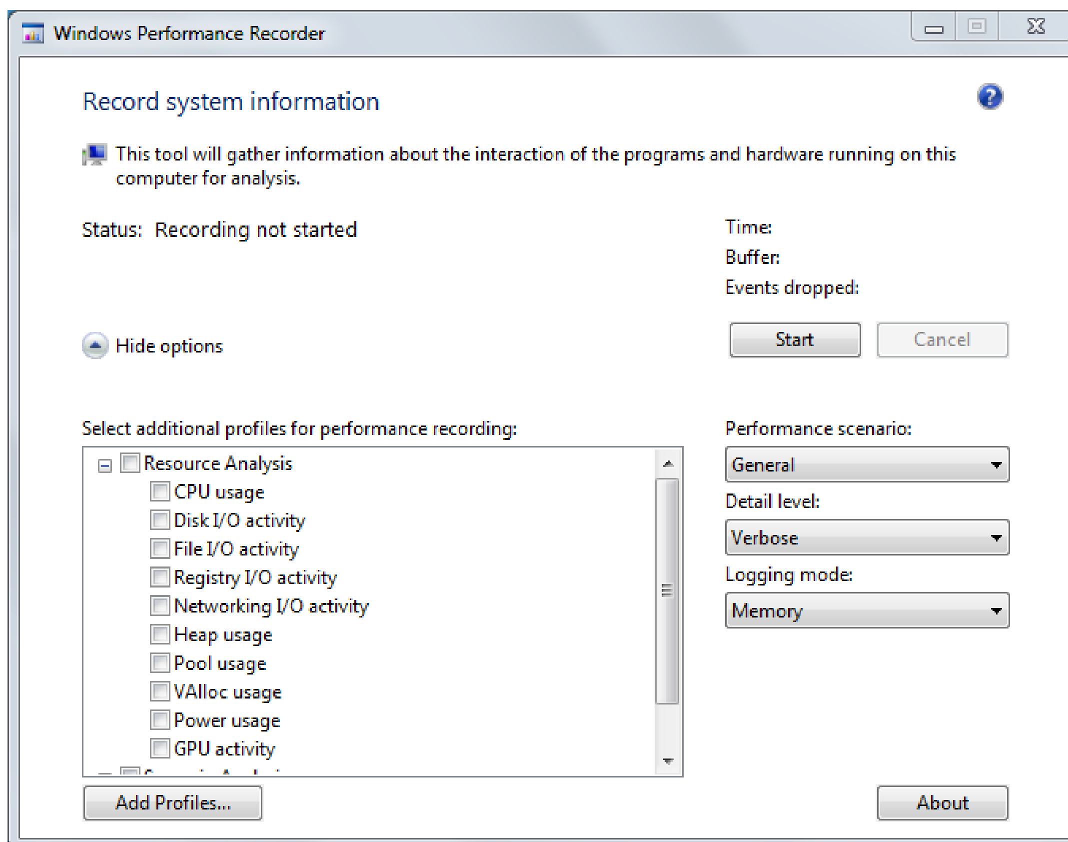
11. ábra Event Tracing for Windows: architektúra

[6] [12] alapján.

3.1.1 ETW vezérlés

A `Windows Performance Recorder` (WPR) egy elsősorban grafikus alkalmazás (bár parancssori hívása is lehetséges), melynek célja az `xperf`, mint ETW vezérlő leváltása. A munkamenet-konfigurációkat saját ún. profilokban tárolva definiálja; jónéhány rendszeresemény-kategóriára a profilok alapértelmezetten elérhetőek.

Figyelem: a memória-bufferek a "Cancel" gomb megnyomására rögtön törlődnek, így a mintákat még a nyomkövetés befejezése előtt mentsük. Javasolt továbbá egyetlen állományt betölteni és egyetlen szűrőt alkalmazni, majd kilépni az alkalmazásból. Ügyeljünk arra is, hogy lehetőleg ne generáljunk többletterhelést a platformon az alkalmazás mellé.



12. ábra Windows Performance Recorder

3.1.2 ETW megjelenítés

A labor során megjelenítésre a `Windows Performance Analyzer-t` (WPA) fogjuk alkalmazni. A WPA `.etl` állományokat értelmező és megjelenítő eszköz.

A 4. ábra a WPR felhasználói felületének alapvető elemeit szemlélteti. A grafikon böngészőből (bal oldalt) húzhatjuk át a nyom különböző esemény-típusainak megjelenítését az analízis-fülekre. A megnyitott nyom egyben példát is mutat arra, hogy miért hasznos az esemény-nyomkövetés alapú teljesítményvizsgálat.

A 5. ábra az adatnézeteket mutatja nagyobb felbontásban. Tegyük fel, hogy azt szeretnénk kideríteni, hogy miért ugrott meg átmenetileg a diszkhasználat (legalsó nézet, „Disk usage – Utilization by Disk, Priority" nézet).

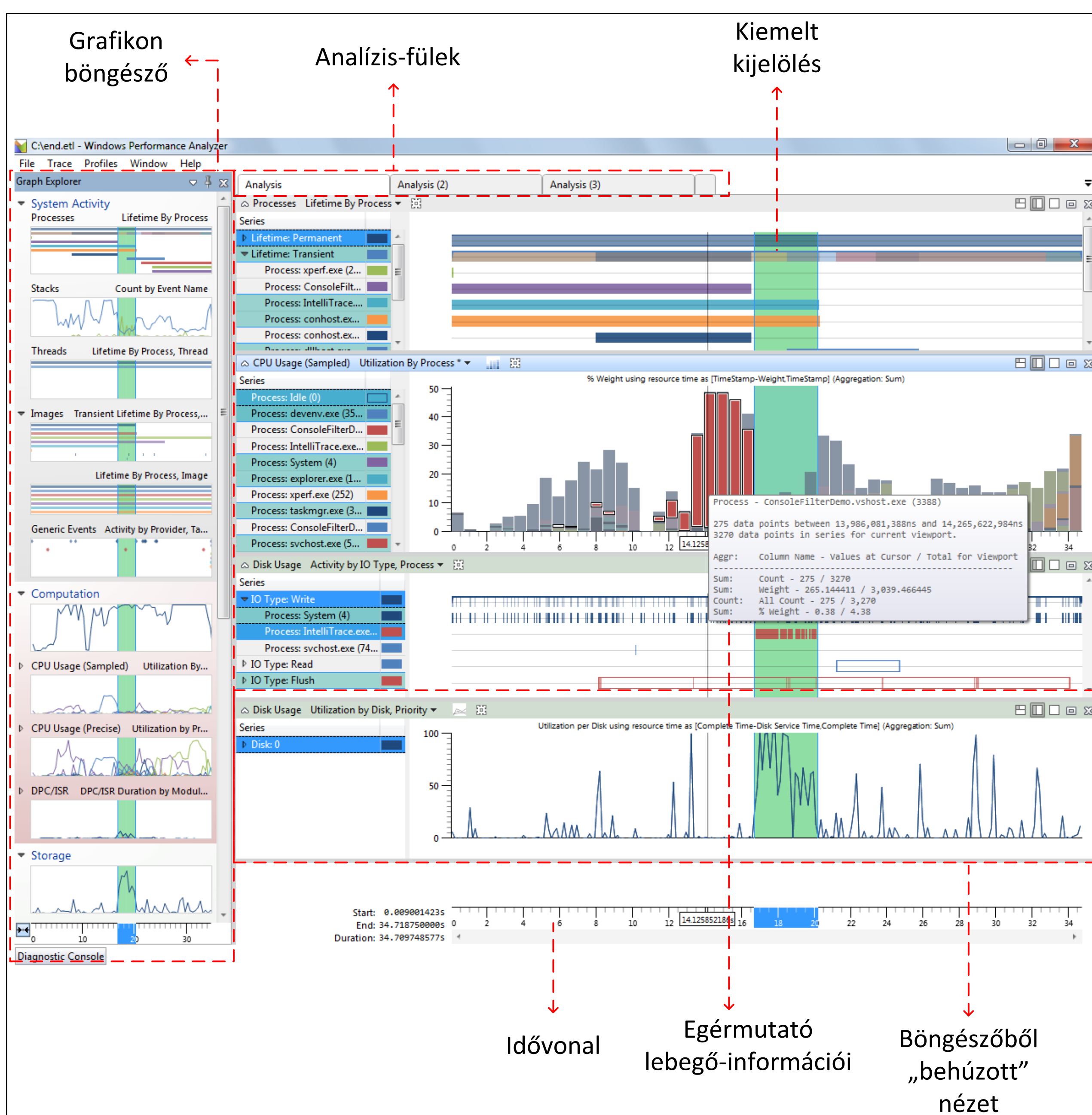
- Először azonosítjuk, hogy ezek a műveletek írások és az `IntelliTrace.exe` végzi őket (harmadik nézet, „Disk Usage – Activity by IO Type, Process".).
- Észleljük, hogy a gyakori írásokat jelentő szakasz egybeesik az `IntelliTrace.exe` futásának végével (legelső nézet).

A kérdést ezzel akár lezártnak is tekinthetjük; az `IntelliTrace` alkalmazás futásának végső fázisa generálta a diszkhasználatot diszk-írásokkal.

Az `IntelliTrace`-ről rövid internetes kereséssel kideríthető, hogy a `Visual Studio` „historikus hibakeresés” funkcióját támogatja. Szokásosan egy debuggerben futás közben tudunk töréspontról töréspontra lépni és a hívási gráfot vagy a helyi változókat vizsgálni; „historikus” támogatással ezt a program futása után, offline is megtehetjük. (Valójában egy alkalmazás-nyomkövetőről van szó ami jelen labornak nem része.)

Ki tudjuk-e deríteni a felvett adatokból, hogy melyik alkalmazás nyomkövetését végezte az `IntelliTrace`? A folyamat-élettörténet és a CPU használat nézetekről már gyanús lehet, hogy a `ConsoleFilterDemo` alkalmazásról van szó, hiszen annak CPU aktivitása (és létezése) pontosan az `IntelliTrace` végső, írási fázisa előtt szűnik meg.

A WPA nézetek konfigurálhatóak; a fejlécben választhatunk grafikus és táblázatos változat között, módosíthatjuk a megjelenített attribútumok halmazát és konfigurálhatjuk a megjelenítést. A 6. ábra a diszkhasználat-események típus és folyamat szerinti bontását mutatja táblázatos formában, a „Path Name” attribútumot bekapcsolva. Ebből a nézetből már egyértelmű, hogy a `ConsoleFilterDemo` nevű, fejlesztés alatt álló alkalmazás nyomkövetésének végső, adatkiírási fázisát figyeltük meg a platform nyomkövetővel.



4. ábra Windows Performance Analyzer

5. ábra Példa: tranziens diszkhasználat okának felderítése

Disk Usage Activity by IO Type, Process *

Line #	IO Type	Process	Path Name	Size
1	Write			25,914,880
2		System (4)		12,508,160
3		IntelliTrace.exe (760)		13,406,208
4			C:\Windows\Temp\ConsoleFilterDemo.vshost.exe_121021_013357_37847db7-4140-412a-9a12-45001406a9d7.iTrace	13,369,344
5			C:\$LogFile	36,864
6		svchost.exe (744)	C:\Windows\ServiceProfiles\LocalService\AppData\Local\lastalive0.dat	512
7	Read			231,424
8	Flush			0

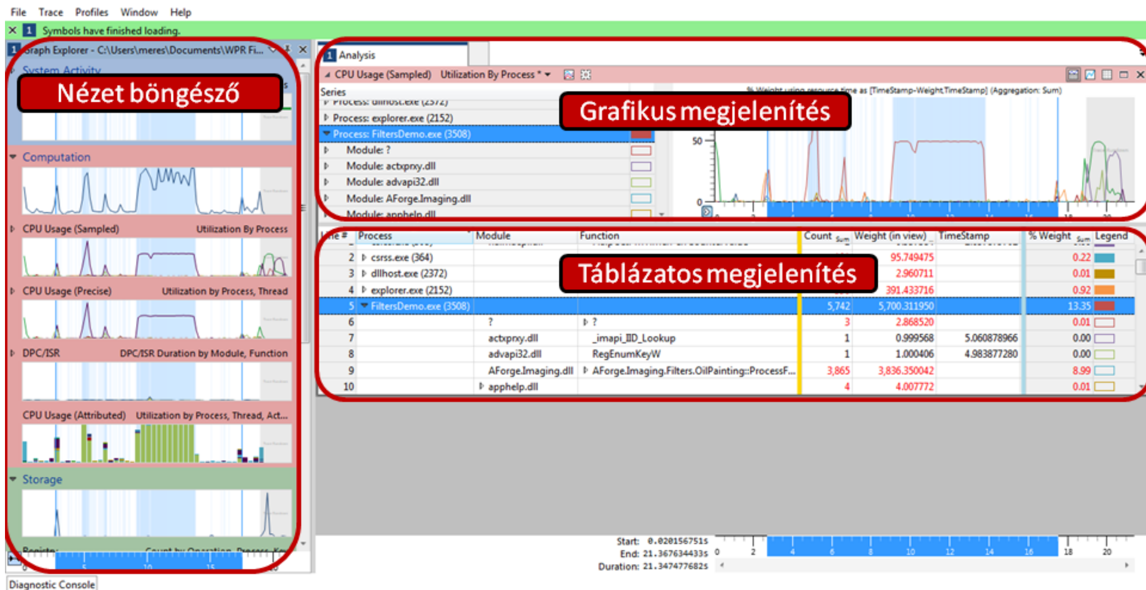
6. ábra Diszkhasználat - táblázatos nézet

A nyomkövetés teljesítmény-diagnosztikára alkalmazására a labor során fogunk több esetet vizsgálni. Az itt bemutatott példa azt hivatott demonstrálni, hogy a nyomkövetés segítségével nem csak az határozható meg nagy pontossággal, hogy mikor, melyik folyamat melyik erőforrást hogyan használta, de sokszor arra is következtetni tudunk, hogy az erőforráshasználat életciklusának mely szakaszához köthető. (Az állapot-megfigyelhetőség egyébiránt felhasználói módú esemény-szolgáltatókkal teljessé tehető.)

Javasolt munkamódszer:

- A nézet-böngésző szintjeinek kibontása és analízis lehetőségek áttekintése.
- Nézetek egymás után analízis-fülre (jobb oldali sávra, „Analysis”) kihúzása és sorban megvizsgálása.

- Grafikus megjelenítés beállítása és értelmezése.
- Táblázatos megjelenítés beállítása és értelmezése (segítséget nyújthat egy későbbi feladat ábrája, a 8. ábra).



7. ábra WPA nézetei

Tippek:

- Táblázatos megjelenítés esetén (8. ábra) a vastag sárga vonaltól balra elhelyezkedő adatokat fa struktúrában lehet kibontani, ami a sárga vonaltól jobbra eső adatok csoportosítását határozza meg. A sárga és kék vonal közötti attribútumok nem jelennek meg a grafikonon, míg a kék vonaltól jobbra elhelyezkedő elemek igen. Azt, hogy mely attribútumok (oszlopok) jelenjenek meg, a „View Editorban" lehet konfigurálni (lásd fogaskerék ikon, 9. ábra). A táblázatban elfoglalt pozíciójuk „drag-and-drop" módszerrel változtatható, a színes elválasztó vonalaktól függetlenül.
- A nézeteknek hasznos tulajdonsága, hogy ha egy objektumot kijelölünk („highlighting"), akkor az összes többi nézetben is a hozzátartozó adatok jelölődnek ki.
- A View Editor (18. ábra) Advanced menüpontját választva szűrőket adhatunk hozzá (pl. Process neve legyen „Profiling.exe" vagy a megnyitott fájl neve tartalmazza azt, hogy „jpg", stb.).
- A veremképet is rögzítő események esetében szükség lehet a rögzített szimbólumok feloldására (például függvényhívások neveinek megjelenítésekor). Ez a menüben a „Trace | Load Symbols" menüpont alatt kapcsolható be. Ez a szimbólumok betöltésére egyrészt helyi (PDB) fájlokat használ fel (amilyet a fordító is elhelyez egy lefordított .NET alkalmazás „Debug" könyvtárába), másrészt távoli szimbólumkiszolgálóktól gyűjtheti össze a szükséges adatokat. Utóbbi eljárás segítségével a Windows binárisok hívásainak többsége is követhető lesz. Jó néhány olyan binárist futtat azonban a virtuális gép, melyekhez nem érhetőek el publikusan szimbólumadatok. A szimbólumok betöltése sok ideig is tarthat, ezért a mérés során érdemes beállítani, hogy csak a mi alkalmazásunk könyvtárában keressen a WPA (Trace | Configure Symbol Paths).
- Senkit ne riasszon el, ha az attribútumok és fogalmak egy jelentékeny része ismeretlennek bizonyul. Az ETW infrastruktúráról tudni kell, hogy azt a Microsoft eredetileg a Windows, mint operációs rendszer belső, operációsrendszer-fejlesztést támogató nyomkövetéséhez kezdte el kifejleszteni. Így az általa nyerhető betekintés mélysége és „szélessége" igen nagy; az összes adat értelmezéséhez a Windows kernel olyan szintű ismerete szükséges, melyet nyilvánvalóan nem követelünk meg.

8. ábra Táblázatos nézet

9. ábra Modulok és függvények hozzáadása a nézethez

4 Párhuzamos programozás

Amennyiben egy IT rendszerben egy adott feladathoz több erőforrásra van szükségünk, alapvetően két megoldás közül választhatunk. Vertikális skálázás esetén a meglévő erőforrásokat nagyobb teljesítményűre cseréljük le (pl.: nagyobb diszk, gyorsabb processzor), horizontális skálázás esetén pedig az erőforrások számát növeljük (pl.: RAID0, többmagos processzor). Mindkét módszer rendelkezik előnyökkel és hátrányokkal is:

- A vertikális skálázás könnyen technológiai és anyagi korlátokba ütközik: előfordulhat, hogy nem gyártanak szükséges teljesítményű diszket, vagy sokkal drágább, mint két feleakkora kapacitású.
- A horizontális skálázódás viszont szoftveres részről nem megy automatikusan: egy hagyományosan megírt program például nem fog gyorsabban futni több processzoron, ha nem készítik fel a párhuzamosság lehetőségeinek kiaknázására.

A párhuzamos programok írása bár elsőre könnyűnek tűnhet, valójában rengeteg kihívást és nehezen észrevehető hibát rejt magában: erőforrások megosztott használata, holtpont, nehezen reprodukálható, nemdeterminisztukus futás. Ezen problémák leküzdésére a keretrendszerek (így a .NET is) számos megoldást kínál.

4.1 Párhuzamos programozás .NET környezetben TPL segítségével

Jelen labor során a horizontális skálázódás teljesítményjellemzőit vizsgáljuk, melyhez a példaprogram a .NET keretrendszer 4.0 verziójától elérhető *Task Parallel Library* (TPL)-re épít. A TPL tulajdonképpen publikus típusok és interfészek (API) gyűjteménye. Célja, hogy könnyebbé tegye a párhuzamos, konkurens alkalmazások fejlesztését. A TPL-ben alapvetően kétféle párhuzamosság létezik:

- *Adatpárhuzamosság* esetén ugyanazt a műveletet kell elvégezni egy nagyobb adathalmaz elemein párhuzamosan. A TPL-ben ilyenkor a `Parallel` osztály `For` és `ForEach` függvényei segítségével a hagyományos `for` és `foreach` ciklusokhoz hasonlóan írhatjuk le az elvégzendő műveletet. A TPL a párhuzamosításról automatikusan gondoskodik, de számos paraméter testre is szabható.
- *Funkcionális párhuzamosság* esetén az elvégzendő munkát kisebb egységekre, úgynevezett taszkokra osztjuk. A taszkok között lehetnek egymástól függetlenek, de sorrendiséget, függőségeket is előírhatunk (pl.: egyik taszk a másik eredményére vár). A TPL-ben a taszk megfelelője a `Task` osztály egy példánya. A keretrendszer automatikusan (de testreszabhatóan) elvégzi a taszkok ütemezését úgy, hogy a lehető legjobb legyen az erőforrások kihasználtsága. Emellett megkönnyíti a megszakítások és kivételek kezelését is.

A labor során az adatpárhuzamossággal foglalkozunk. Az alábbiakban néhány egyszerű példával szemléltetjük a `Parallel.For` és `ForEach` működését.

```
// Soros
for (int i = 0; i < 10; i++)
{
    Console.WriteLine("{0}", i);
}

// Párhuzamos
Parallel.For(0, 10, i => Console.WriteLine("{0}", i));
```

A `Parallel.For` legegyszerűbb változatának (többféle túlterhelés is létezik) három paramétere van. Az első kettő a ciklus kezdő és végértéke, míg a harmadik egy lambda kifejezés, amely tulajdonképpen egy anonim függvény. A kifejezésben a `=>` operátortól balra találhatóak a bemenő paraméterek (jelen esetben egyetlen változó), jobb oldalt pedig a függvény törzse (jelen esetben egyetlen utasítás, de kapcsos zárójelek közé zárva több utasítás is írható).

```
// Soros
foreach (var x in collection)
{
    Console.WriteLine("{0}", x);
}

// Párhuzamos
Parallel.ForEach(collection, x => Console.WriteLine("{0}", x));
```

A `Parallel.ForEach` legegyszerűbb változatának (itt is több túlterhelés létezik) első paramétere a kollekció, a második pedig egy lambda kifejezés, amelyet az első paraméter elemein kell párhuzamosan végrehajtani.

Az előző két egyszerű példánál szinte mindent a keretrendszerre bízunk. Automatikusan eldönti, hogy mekkora részekre legyen feldarabolva a kollekció a párhuzamos végrehajtáshoz és azt is, hogy hány szálon történjen a végrehajtás. Ezek természetesen mind széleskörűen testreszabhatóak, a példaprogramban utóbbira láthatunk is demonstrációt.

5 Hivatkozások

[1] J.F. Meyer. On Evaluating the Performability of Degradable Computing Systems. *IEEE Transactions on Computers*, C-29(8):720-731, 1980, IEEE Computer Society.

[2] dr. Majzik István, Micskei Zoltán. Futásidő, memóriahasználat monitorozása (profiling). *A „Szoftverellenőrzési technikák" tárgy oktatási segédanyaga.* http://www.inf.mit.bme.hu/sites/default/files/materials/category/kateg%C3%B3ria/oktat%C3%A1s/msc-t%C3%A1rgyak/szoftverellen%C5%91rz%C3%A9si-technik%C3%A1k/11/SZET-2011-EA11b_profiling.pdf

[3] Dmitry Evdokimov, Digital Security Research Group. Light and Dark side of Code Instrumentation. *CONFidence 2012*, Krakkó, 2012.

[4] MSDN. How to: Use a Symbol Server. http://support.microsoft.com/kb/319037

[5] Wikipedia. „Program database" szócikk. http://en.wikipedia.org/wiki/Program_database

[6] Wikipedia. „Debug symbol" szócikk. http://en.wikipedia.org/wiki/Debug_symbol

[7] Wikipedia. „Interrupt storm" szócikk. http://en.wikipedia.org/wiki/Interrupt_storm

[8] MSDN. Debugging an Interrupt Storm. http://msdn.microsoft.com/en-us/library/windows/hardware/ff540586(v=vs.85).aspx

[9] MSDN. How to: Install the Stand-Alone Profiler. http://msdn.microsoft.com/en-us/library/bb385771.aspx

[10] SourceWare. SystemTap projekt honlapja, http://sourceware.org/systemtap/

[11] D.A. Reed. Performance Instrumentation Techniques for Parallel Systems. *Lecture Notes in Computer Science*, 729:469-490, 1993, Springer, doi: 10.1007/BFb0013864

[12] T. Soulami. Inside Windows Debugging – A Practical Guide to Debugging and Tracing Strategies in Windows. 2012, Microsoft Press, ISBN: 978-0-7356-6278-0

[13] Dr. Insung Park, Ricky Buch. Improve Debugging And Performance Tuning With ETW. http://msdn.microsoft.com/en-us/magazine/cc163437.aspx

[14] „Windows 7 Instrumentation and Performance" – Windows 7 oktatóanyag. http://download.microsoft.com/download/8/C/D/8CD015BB-081B-49C5-A506-9C9B570B8DD2/InstrumentationAndPerformance.pptx

[15] Wikipedia. „List of of performance analysis tools" szócikk. http://en.wikipedia.org/wiki/List_of_performance_analysis_tools

[16] MSDN. Analyzing Application Performance by Using Profiling Tools. http://msdn.microsoft.com/en-us/library/z9z62c29.aspx

[17] Az MSDN Channel 9 közösségi oldala. „Visual Studio Toolbox: Performance Profiling" videobejegyzés. http://channel9.msdn.com/Shows/Visual-Studio-Toolbox/Visual-Studio-Toolbox-Performance-Profiling

[18] MSDN. Task Parallel Library (TPL), http://msdn.microsoft.com/en-us/library/dd460717%28v=vs.110%29.aspx