Informatikai technológiák laboratórium 2

Teljesítményjellemzők vizsgálata

Mérési segédlet

Készítette: Kocsis Imre

ikocsis@mit.bme.hu

2013.

Verzió: 1.0

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. Egyértelmű kapcsolatuk miatt – pl. egy nem megfelelően méretezett rendszer túlterhelése annak rendelkezésre nem állásához vezethet, vagy a rendszer belső hibái a teljesítmény csökkenéséhez – az angol szakterminológiában a szolgáltatásbiztonságnak (*dependability*) a teljesítménnyel együtt kezelt más aspektusaira szokás *„performability*” („teljesítőképesség”) néven is hivatkozni.

Az IT rendszerek futásidejű teljesítménye szokásosan igen sok faktortól függ; a rendszertervezés külön szakterülete a kapacitástervezés (*capacity planning*), melynek célja a rendelkezésre álló erőforrások olyan méretezése, hogy azok a jelenlegi és jövőbeli üzleti igényeket költséghatékony módon kielégítsék. Itt az üzleti igények részének tekintendő a szolgáltatásokkal szembeni teljesítménykövetelmények megfogalmazása is. „Az elvárt teljesítményt” leíró rendszeraspektus és az azt számszerűsítő metrika rendszertől, alkalmazástól függően persze igen változékony. Míg grafikus asztali alkalmazások esetén leginkább a „túl lassan fut” jellegű hibabejelentésekkel találkozhat a rendszeroperátor és a fejlesztő, addig a kellően fontos feladatot ellátó elosztott kiszolgálórendszerek esetén nem csak a szolgáltatási szintű teljesítményt leíró metrikákat szokás precízen definiálni, de azt és a szolgáltatást megvalósító platformok teljesítmény-mérőszámait (pl. CPU vagy memória-kihasználtság) folyamatosan monitorozni is.

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 nem megfelelő teljesítményben megnyilvánuló hibahatást kiváltó hibaokokat a szoftver- és rendszertervezés, majd az implementáció és a tesztelés során törekszünk eltávolítani, amennyiben lehetséges (ilyenek pl. a szoftver-, méretezési és konfigurációs hibák). A megfelelő teljesítmény biztosításához kapcsolódó, a rendszerfejlesztés során jellemzően végrehajtott tevékenyégek igen sokrétűek lehetnek; nem csak a feljesztés alatt álló rendszer, de a fejlesztési folyamat is befolyásolja őket. Mindemellett elmondható, hogy 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 szoftver-teljesítményjellemzők méréseken alapuló vizsgálata egy rendszer életciklusában résztvevő több szereplőnek a feladata is lehet.

1. A *szoftverfejlesztő* nézete jellemzően lokális, a fejlesztett és a fejlesztettel közvetlen kölcsönhatásban álló szoftvermodulokra koncentrál. A teljesítményjellemzők megfigyelése általában igen finom felbontású (pl. program hívási gráfja és a függvényekben eltöltött idő).

2. A *rendszertervező* és *rendszerintegrátor* az alkalmazás integrációs teljesítménytesztelése során a teljes rendszer kontextusát vizsgálja durvább felbontásban, mint a fejlesztő. A fejlesztéssel szemben a hangsúly

    a. a rendszerkomponensek által nyújtott szolgáltatások szolgáltatás-minőségének

    b. és a rendszer által végzett absztrakt feladatok (pl. elosztott tranzakciók) végrehajtási teljesítményjellemzőinek

    mérésén van. Itt az alkalmazást kevésbé mélyen szükséges megfigyelni – pl. modulokon belüli belső hívási gráfra nincs szükség, már csak azért sem, mert a rendszert tesztelők nem feltétlenül ismerik a feldolgozási logikát ilyen mélységben. Szükséges azonban a *komponensek feladatvégrehajtásának* (pl. egy alkalmazásszerverben a kéréskiszolgálás főbb lépései), az *elosztott feladatvégrehajtásnak* (pl. több architekturális rétegen átívelő tranzakciók) és ezek minőségének megfigyelhetősége. Az ehhez elengedhetetlen alkalmazás-instrumentáció legtöbbször jellegéből adódóan nem lehet teljesen automatikus, így tervezési és fejlesztési időráfordítást igényel.

    Ezen a szinten arra is megjelenik az igény, hogy a platform működését együtt vizsgálhassuk az alkalmazás(ok)éval például a szűk keresztmetszetek azonosítása érdekében. Ehhez a jelen segédlet által is tárgyalt *platform-eseménynyomkövetés* és a működtetés közbeni platform-monitorozást lehetővé tevő *teljesítményszámlálók* a legfontosabb eszközeink.

3. A *rendszeroperátor* a működő rendszer teljesítményét követi és

    a. az abban észlelt zavarok okát kívánja meghatározni,

    b. a hibás állapotot helyreállítani és

    c. a hibaokokat eltávolítani.

    A hibás szoftver-logikából illetve szoftver-integrációból adódó hibaokokat operátori szinten persze nem lehet eltávolítani, csak a keletkezett hibás állapotot helyreállítani (pl. újraindítással); a fizikai hibákat és a túlterheléseket azonban a rendszermenedzsment feladata megszüntetni.

    A működés közbeni teljesítmény-megfigyeléshez durvább felbontású megközelítéseket alkalmazunk, pl. erősen szelektív és csak a problémákat jelző naplózást és a teljesítményszámlálók monitorozását. Az ismétlődő, illetve reprodukálható hibák esetén azonban itt is szükség lehet finomabb felbontású eszközökre a hibabehatároláshoz; pl. platform-eseménynyomkövetés segíthet annak eldöntésében, hogy a teljesítmény-hibahatás oka az alkalmazási logikában vagy a rendszerintegrációban keresendő (ezeket nem az üzemeltetés tudja orvosolni), vagy platform szinten.

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 és az *esemény-nyomkövetéssel* (*event tracing*). Szándékunk szerint ez a párosítás a szakirány mindhárom ágazatának hallgatói számára specializációjukhoz is jól illeszkedő ismereteket ad át.

A *Teljesítményjellemzők vizsgálata* c. mérésre való felkészüléshez ez a segédlet bemutatja a profiling megközelítéshez (2. fejezet) a Visual Studio 2012 Ultimate eszköz működését (3. fejezet), majd az esemény-nyomkövetéshez (4. fejezet) az Event Tracing for Windows infrastruktúrát (5. fejezet).

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 legnagyobb 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).

A „profiling"-ként annotált programanalízis-megközelítésekre általában igaz az továbbá – különösen a futásidő-profiling esetén –, hogy egy futási „profil" a megfigyelésekre nem önálló eseményként tekint, hanem azok csak egy „számolt" statisztikai jellemző értékét befolyásolják[2]. Így a futásidő-profiling esetén egy függvénybe belépés vagy abból visszatérés (vagy annak megfigyelése, hogy a processzor programszámlálója egy adott függvényhez tartozó címtartományon belül tartózkodott) nem önálló események, hanem pl. „az adott függvény végrehajtásával töltött összes idő" metrika értékéhez járulnak hozzá.

Egy futó program profiling célú, szoftveres úton való megfigyelésére alapvetően két lehetőségünk kínálkozik; a mintavételezés és az instrumentáció. (A hardver alapú megközelítésekkel itt szándékoltan nem foglalkozunk.) Ezeket a futásidő-profilingon keresztül vezetjük be.

2.1 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 egyrészt jellemzően nem igénylik a kód/futtatókörnyezet felműszerezését, másrészt hatásuk a rendszer futási teljesítményére alacsony és egyértelműen szabályozható.

2.2 Felműszerezés

A futtatott kód *felműszerezése* (*instrumentation*) során olyan extra utasítások kerülnek beszúrásra és végrehajtásra, melyek a végzett műveletekről képesek a profiler eszközt „tájékoztatni". Ez

[1] [2] felhasználásával.

[2] Az óvatos fogalmazás oka, hogy a „profiling" mára nagyon általános fogalommá vált, igen széles eszközkészlettel. Így egyes alkategóriái esetén a közös jellemzők ugyan rendre jól körülírhatóak, de ezek mellett a modern eszközök további, sokszor pl. esemény-nyomkövetést is magukban foglaló funkciókat is nyújthatnak.

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[3]. A felműszerezés – „extra” hívások beszúrása – a következő szinteken történhet ([3] alapján):

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.

2.3 Megjelenítés

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*) kerülnek előállításra és megjelenítésre.

2.3.1 Call tree

A hívási fa a metódusokban/függvényekben eltöltött idő top-down bontását adja meg. Az 1. ábra a labor során használt `Visual Studio 2012 Ultimate` profilere egy példa-futására mutat egy hívási fa megjelenítést[4].

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 szummázva 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:

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

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

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[5].
3. Figyeljük meg, hogy a `System.Console.WriteLine` hívás két szinten is megjelenik (kétszer 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.

2.3.2 Flat profile

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.Drawin	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
System.Console.WriteLine(string)	4	0.07	0.07	0.39	0.39
ConsoleFilterDemo.ConsoleFilterEventSource..cctor()	1	0.75	0.03	16.84	0.71
ConsoleFilterDemo.ConsoleFilterEventSource.ImageLoa	1	0.02	0.02	0.46	0.39

2. ábra. „Flat profile" példa.

2.3.3 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 nagy számú és nem blokkolt folyamat/szál teljesítményproblémákhoz 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.

[5] 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].

Érdemes már itt megjegyeznünk, hogy az erőforrás-kölcsönhatások által okozott problémák felderítése részben túlmutat a profiling legegyszerűbb használati esetén, amikor azt fejlesztés (illetve teljesítmény-tesztelés) közben alkalmazzuk, a fejlesztett programra koncentrálva.

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*)[6] 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.

Problémát az okoz, hogy az ilyen teljesítmény-hibák kiváltó okának felderítéséhez egyidejűleg szükséges a *(külső) hibaok aktív fennállása/aktivációja* és a futtató *platform megfigyelhetősége*.

- **Hibaok-aktiváció.** A külső hibaokokkal jellemzően működtetés és nem tesztelés közben szembesül egy alkalmazás, hacsak fejlesztés közben nem vetették alá célzott hibainjektálási kísérleteknek.
- **Platform-megfigyelhetőség.** A pontos diagnosztikához szükségünk lehet olyan platformszintű vagy más alkalmazásokból származó eseményekre és mérőszámokra (pl. folyamatonkénti CPU-használat, megszakítások kiszolgálásával töltött idő, diszk írások és olvasások statisztikái), melyeket önmagában a szigorúan értelmezett profiler nem vizsgál.

Az első problémára részben megoldást ad az, hogy a profilereket jellemzően fejlesztői környezet nélkül is lehet futtatni[7]. Persze ekkor is csak ismételten jelentkező hibák esetén *érdemes* használni őket, mivel a folyamatos profiling erőforrásigénye – minimálisan a regisztrátumok tárolásának és feldolgozásának szükségessége miatt – ha alacsony is, nem elhangyagolható.

A platformkörnyezet eseményeinek és az alkalmazásoknak a platformkörnyezettel való interakciójának megfigyelését általában a platformszintű *esemény-nyomkövető* vagy a monitorozást lehetővé tevő teljesítményszámláló alrendszerekkel valósíthatjuk meg, nem pedig a profiling technológiák segítségével. (Modern Windows platformokon ezek az `Event Tracing for Windows` – ETW – és a `Windows Performance Counters` technológiák.) Egy profiler opcionálisan ezek segítségével csatolhat esemény-megfigyeléseket a profiling mérések mellé (mint arra a `Visual Studio 2012` képes is). Emellett ha a platform esemény-nyomkövető rendelkezik pl. időzített CPU mintavételezési eseménnyel és a mintákhoz képes stack pillanatképeket is csatolni, akkor a megfigyelések alkalmazhatóak lesznek rendszerszintű (több folyamatot és a kernelt is magába foglaló) profilingra is. Ezt a funkcionalitást az ETW tárgyalásánál látjuk majd, de hasonló képességekkel rendelkezik Linux környezetben pl. a SystemTap [10] infrastruktúra is.

[6] A mérésen túlmutató, de javasolt olvasmányok: [7] és [8] (a debugger használatát leíró részig).

[7] `Microsoft Visual Studio 2012` esetében lásd: [9].

2.4 Egyéb dinamikus programaspektusok

A futásidő-profiling mellett a profiler eszközök szokásosan lehetővé teszik a memória-foglalások és felszabadítások, valamint egy program konkurencia-viszonyainak megfigyelését és analízisét is. Jelen segédletben ezekre külön nem térünk ki.

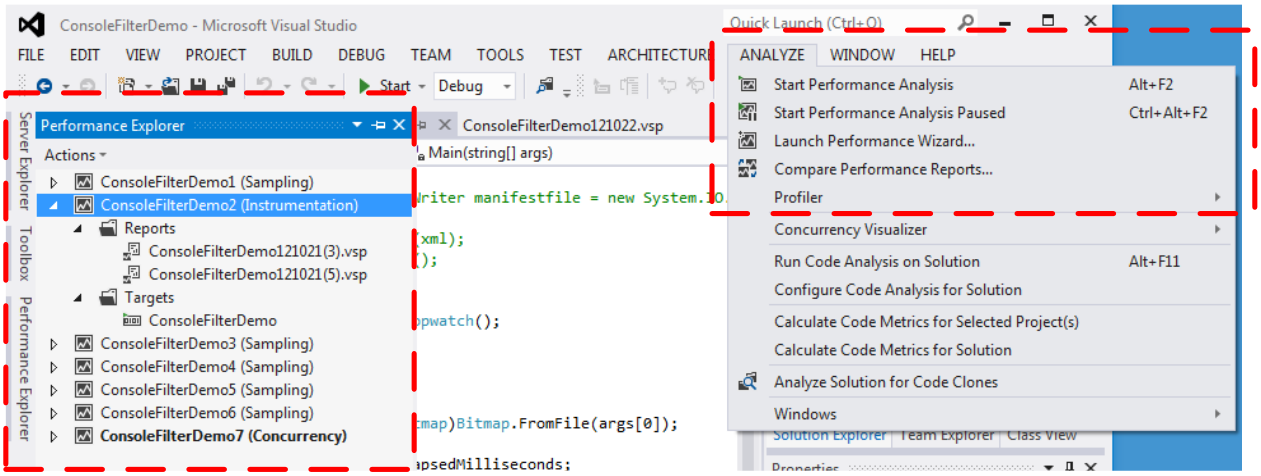
2.5 Profiler technológiák

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 2012 Ultimate` segítségével ad bevezetést az (alkalmazás) profiling mint általános megközelítés területére.

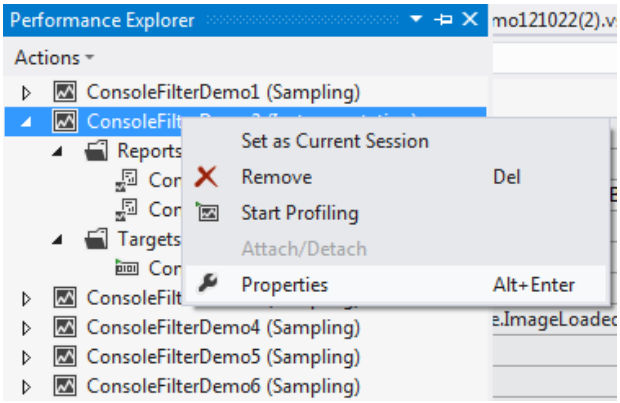
3 .NET profiling a Visual Studio 2012 Ultimate segítségével

A `Visual Studio 2010` és `2012 Premium` és `Ultimate` verziói beépített és az IDE-vel integrált profiling támogatással rendelkeznek.

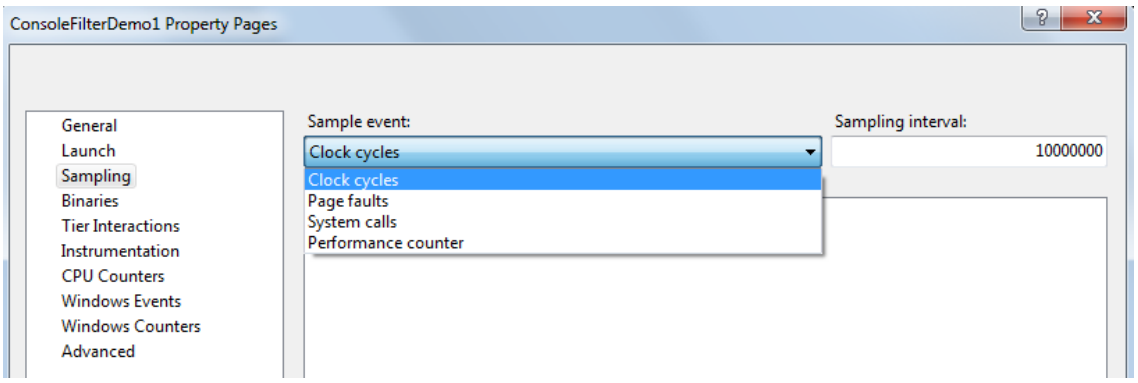


3. ábra. Visual Studio: profiling akciók és munkamenet-böngésző.

Egy (Visual C#) projektre megnyitott fejlesztőeszközben az „`ANALYZE`" menüpontból érhetőek el a projekten (illetve az aktuális profiling munkameneten) értelmezett profiling akciók. A definiált munkameneteket, azok futásának jelentéseit és a célpontjait a „`Performance Explorer`" segítségével nyithatjuk meg (lásd 3. ábra). A munkamenet-böngésző elemeinek kontextusmenüivel módosíthatjuk is az elemeket, és azokon értelmezett akciókat indíthatunk el (4. ábra és 5. ábra).

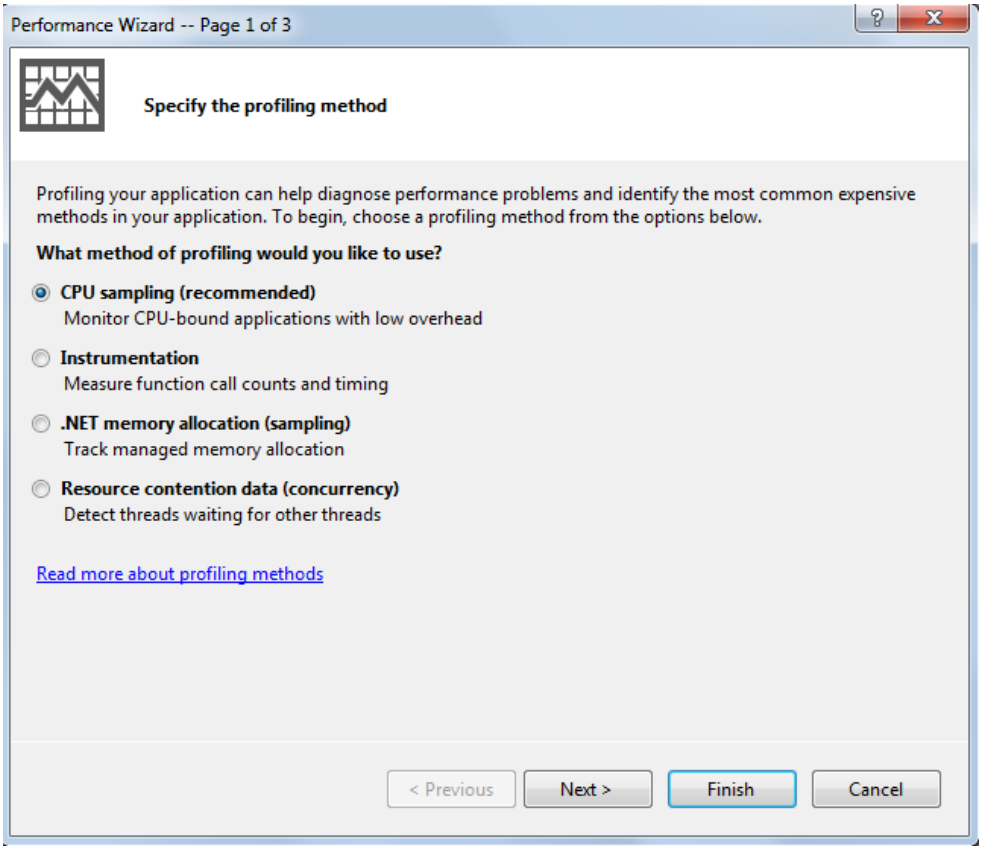


4. ábra. Munkamenet-kontextusmenü.



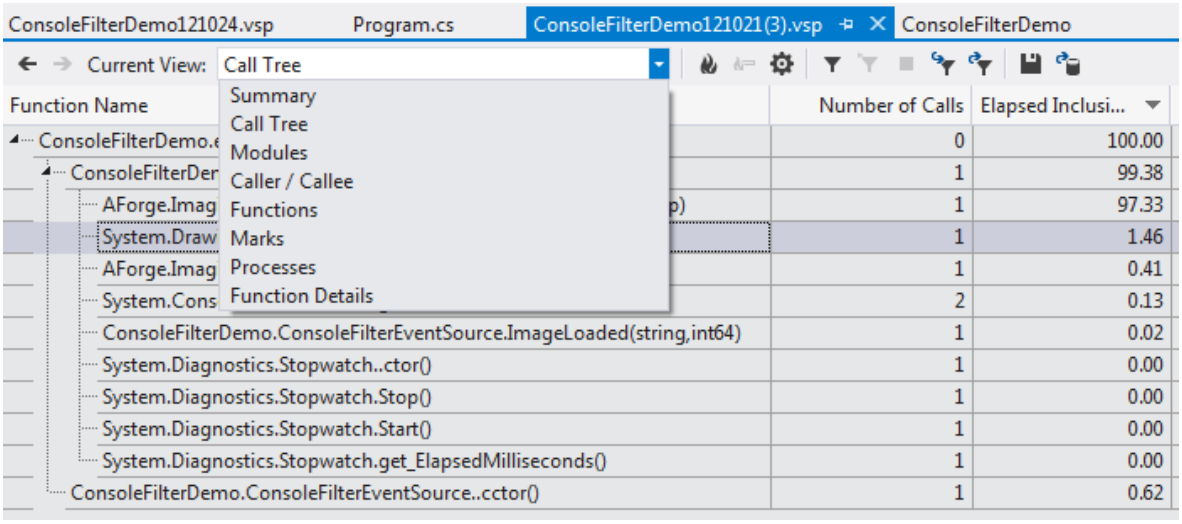
5. ábra. Mintavételező profiling munkamenet (sampling) tulajdonságai.

Az új profiling munkamenetek definiálásának és indításának legegyszerűbb eszköze a „teljesítmény varázsló” (`ANALYZE → Launch Performance Wizard...`, 6. ábra).



6. ábra. A „teljesítmény varázsló”.

Az elkészült teljesítményjelentéseket önálló lapokon nyithatjuk meg, melyeken belül különböző megjelenítési nézetek érhetőek el (7. ábra).



7. ábra. Teljesítményjelentés megjelenítési nézetei.

A `Visual Studio` profiling eszközök használata a már korábban bevezetett alapfogalmak ismeretében meglehetősen intuitív, így mélyebben nem tárgyaljuk azt.

A mérésre való felkészülés részeként mindemellett javasolt az MSDN *„Analyzing Application Performance by Using Profiling Tools”* [16] lapjának áttekintése abból a célból, hogy a mérés során a lap struktúrája már ismert legyen, és on-line referenciaként tudjon szolgálni. Igen jó „hands on” stílusú bevezetést ad még a Channel 9 „Visual Studio Toolbox: Performance Profiling” bejegyzése [17]. A videó hossza miatt annak megtekintését nem várjuk el, csak javasoljuk.

4 Esemény-nyomkövetés[8]

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 rögzített események ezen aktivitások példány-reprezentációi, melyek tartalmaznak esemény-attribútumokat is. 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. (Az olvasóban felmerülhet a kérdés, hogy egy instrumentáló profiler valójában profiler vagy esemény-nyomkövető. A segédlet által támogatott mérés szempontjából a válasz az, hogy profiler,

[8] Részben [11] 3.3 alfejezete alapján.

mivel kimenetei segítségével aggregált „profilokat” és nem esemény-szekvenciákat vizsgálhatunk.) A profiling nyomkövetésre visszavezetésére egy másik módot adnak a korábban említett „mintavételező” események.

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.

Valójában az alkalmazási szintű esemény-nyomkövetés felfogható a naplózás egy specializált eseteként is. A legfőbb különbség az, hogy a normál működés közben folyamatosan végzett és általában a rendszeroperátorokat megcélzó eseménynaplózás magasabb szintű (így jellemzően sokkal ritkább) eseményeket generál, mint a forráskódon belüli hibaizolációt is lehetővé tevő nyomkövetés, melynek fő célközönsége így a fejlesztők.

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 Event Tracing for Windows[9]

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 felhasználói módú folyamatokban allokált adatstruktúrák helyett a buffereket a Windows kernel kezeli. A naplózás zármentes (*lock-free*); a buffereket a kernel üríti diszkre és használja újra. A diszkre írást dedikált kernel-szálak végzik, így

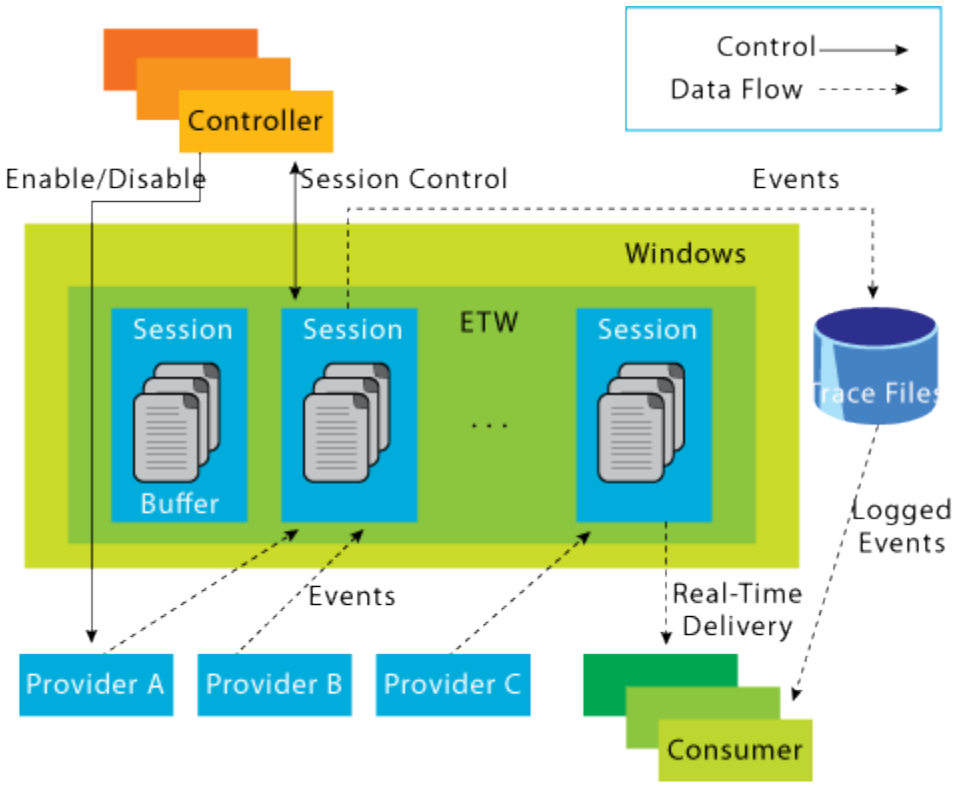
[9] [12] alapján.

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 buffereket 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 8. ábra szemlélteti a komponensek kapcsolatait.



8. ábra. Event Tracing for Windows: architektúra [13]

Az ETW instrumentáció mind a kernelben, mind pedig a nyomkövetésre felkészített alkalmazásokban állandóan jelen van, de csak nyomkövetés közben aktív; alkalmazások esetén ez azt jelenti, hogy nyomkövetési célokra nem szükséges speciálisan fordított programváltozatok telepítése, csak az esemény-forrásokat kell „bekapcsolni”.

A nyomkövetés és a naplózás hasonlóságát felismerve a modern Windows platformokon az ETW és az „Event Log” infrastruktúra nagymértékben harmonizált; API-halmazuk és fejlesztési modelljük között igen nagy az átfedés. Az 1. táblázat a két technológia néhány fontosabb jellemzőjét hasonlítja össze.

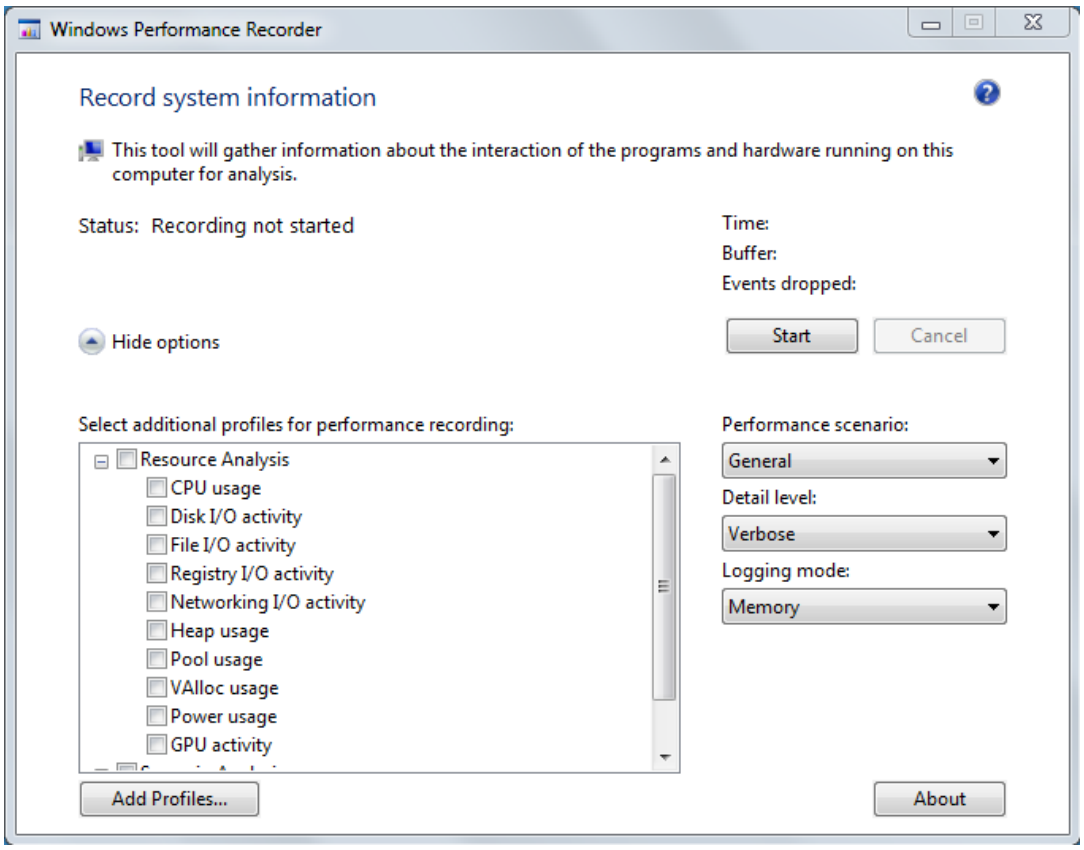
1. táblázat. Az ETW és a Windows Event Log összehasonlítása. [14]

Célközönség	**Fejlesztő**	**Rendszeradminisztrátor**
Technológia	ETW	Event Log
Eseményvezérlés	szelektív (munkamenetek indításával)	„Always On”
Esemény-ráta	magas (10^4/s)	közepes (10^2 – 10^4/s)
Jellemzők	Deklaratív definíció manifest-ekben Feldolgozási (fogyasztó) API Rendelkezésre álló szolgáltatók felfedezhetősége Flexibilis adatmodell	Mint az ETW-nél + távoli begyűjtés lekérdezés-támogatás központosított naplók

5.1 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.

A mérés során mi azonban a WPR helyett részben az `xperf`-et fogjuk használni. Ennek fő okai, hogy a) az `xperf` jónéhány olyan tevékenységet támogat, amit a WPR nem (pl. szolgáltatók enumerálása); b) a profil bázisú konfigurációs stratégia mellé egyelőre nem létezik profil-szerkesztőeszköz, így a sémahelyességen túl a WPR igényeinek valóban megfelelő XML profildefiníciók szerkesztése igencsak nehézkes.



9. ábra. Windows Performance Recorder.

5.1.1 Munkamenet indítása

Az `xperf` eszközben a következő szintaxissal indíthatunk munkameneteket:

```
xperf.exe –start [<munkamenet_nev>] –on <Szolgaltato_1+ Szolgaltato_2+...+ Szolgaltato_n>
```

Az `xperf` hívásokat adminisztrátorként indított `cmd.exe`-ből végezzük. A munkamenet-név elhagyása esetén egy egy speciális, `NT Kernel Logger` nevű munkamenet kerül elindításra; ez azonban csak kernel módú szolgáltatókat tud indítani. Névvel hivatkozott munkamenet indításához viszont a `Kernel Loggernek` már futnia kell. A munkamenetek teljes konfigurációját (pl. buffer-méret) meghatározó további kapcsolókra nem térünk ki; azok az

```
xperf –help start
```

hívással felsoroltathatóak. A további help-kategóriákat egyszerűen az

```
xperf –help
```

hívással jeleníthetjük meg.

5.1.2 Szolgáltatók lekérdezése

A rendelkezésre álló szolgáltatók lekérdezése pl. a

```
xperf –providers [Installed|I] [Registered|R] [KernelFlags|KF] [KernelGroups|KG]
```

paranccsal történhet.

I: egy *felhasználói módú* szolgáltató akkor „installált”, ha leíró manifest-je (ami pl. az eseményeket vagy az esemény emittáló binárisokat adja meg) installálva lett (pl. a `wevtutil` parancssori eszközzel). Másszóval ezek azok a nem-kernel szolgáltatók, melyek a platformon jelen vannak.

R: egy *felhasználói módú* szolgáltató akkor „regisztrált”, ha van olyan futó folyamat, mely az ő eseményeit emittálhatja. (Ez nincs összefüggésben azzal, hogy a szolgáltatóra fut-e már nyomkövetés.)

KF: A `KernelFlags` opció a kernel módú szolgáltatókat sorolja fel. Egy példa-futás (Windows 7):

```
PROC_THREAD    : Process and Thread create/delete
LOADER         : Kernel and user mode Image Load/Unload events
PROFILE        : CPU Sample profile
CSWITCH        : Context Switch
COMPACT_CSWITCH: Compact Context Switch
DISPATCHER     : CPU Scheduler
DPC            : DPC Events
INTERRUPT      : Interrupt events
SYSCALL        : System calls
PRIORITY       : Priority change events
SPINLOCK       : Spinlock Collisions
KQUEUE         : Kernel Queue Enqueue/Dequeue
ALPC           : Advanced Local Procedure Call
PERF_COUNTER   : Process Perf Counters
DISK_IO        : Disk I/O
DISK_IO_INIT   : Disk I/O Initiation
FILE_IO        : File system operation end times and results
FILE_IO_INIT   : File system operation (create/open/close/read/write)
HARD_FAULTS    : Hard Page Faults
FILENAME       : FileName (e.g., FileName create/delete/rundown)
SPLIT_IO       : Split I/O
REGISTRY       : Registry tracing
REG_HIVE       : Registry hive tracing
DRIVERS        : Driver events
POWER          : Power management events
CC             : Cache manager events
NETWORKTRACE   : Network events (e.g., tcp/udp send/receive)
VIRT_ALLOC     : Virtual allocation reserve and release
MEMINFO        : Memory List Info
ALL_FAULTS     : All page faults including hard, Copy on write, demand zero faults, etc.
MEMINFO_WS     : Working set Info
VAMAP          : MapFile info
FOOTPRINT      : Support footprint analysis
MEMORY         : Memory tracing
REFSET         : Support footprint analysis
CONTMEMGEN     : Contiguous Memory Generation
POOL           : Pool tracing
CPU_CONFIG     : NUMA topology, Processor Group and Processor Index to Number mapping. By default
it is always enabled.
SESSION        : Session rundown/create/delete events.
IDLE_STATES    : CPU Idle States
TIMER          : Timer settings and its expiration
CLOCKINT       : Clock Interrupt Events
IPI            : Inter-processor Interrupt Events
OPTICAL_IO     : Optical I/O
OPTICAL_IO_INIT: Optical I/O Initiation
FLT_IO_INIT    : Minifilter callback initiation
FLT_IO         : Minifilter callback completion
FLT_FASTIO     : Minifilter fastio callback completion
FLT_IO_FAILURE : Minifilter callback completion with failure
HAL_CLOCK      : HAL Clock Configuration events
```

A kernel-flagekkel általában nem közvetlenül dolgozunk, hanem csoportosítva adjuk meg őket.

KG: a kernel módú szolgáltatókat az ETW a könnyebb alkalmazhatóság végett csoportokba sorolja; a csoportok neve használható a csoportot alkotó szolgáltatók felsorolása helyett. A KG opció ezeket íratja ki.

```
Base           : PROC_THREAD+LOADER+DISK_IO+HARD_FAULTS+PROFILE+MEMINFO+MEMINFO_WS
Diag           : PROC_THREAD+LOADER+DISK_IO+HARD_FAULTS+DPC+INTERRUPT+CSWITCH+
                 PERF_COUNTER+COMPACT_CSWITCH
DiagEasy       : PROC_THREAD+LOADER+DISK_IO+HARD_FAULTS+DPC+INTERRUPT+CSWITCH+PERF_COUNTER
Latency        : PROC_THREAD+LOADER+DISK_IO+HARD_FAULTS+DPC+INTERRUPT+CSWITCH+PROFILE
FileIO         : PROC_THREAD+LOADER+DISK_IO+HARD_FAULTS+FILE_IO+FILE_IO_INIT
IOTrace        : PROC_THREAD+LOADER+DISK_IO+HARD_FAULTS+CSWITCH
ResumeTrace    : PROC_THREAD+LOADER+DISK_IO+HARD_FAULTS+PROFILE+POWER
```

```
SysProf         : PROC_THREAD+LOADER+PROFILE
ResidentSet     : PROC_THREAD+LOADER+DISK_IO+HARD_FAULTS+MEMORY+MEMINFO+VAMAP+SESSION+
                  VIRT_ALLOC
ReferenceSet    : PROC_THREAD+LOADER+HARD_FAULTS+MEMORY+FOOTPRINT+VIRT_ALLOC+MEMINFO+VAMAP+
                  SESSION+REFSET+MEMINFO_WS
Network         : PROC_THREAD+LOADER+NETWORKTRACE
```

5.1.3 Stackwalk

Bizonyos kernel-események esetén értelmezhető és potenciálisan fontos az eseményhez kapcsolható stack-pillanatkép. Ezeknek az eseményekkel együttes gyűjtését az xperf a `–stackwalk` flag segítségével teszi lehetővé. Példa erre a `CPU Sample Profile`, mely a `Profile` eseményre stackwalk segítségével mintavételezett profilingot valósít meg; de a rendszerhívások (`SYSCALL`) belépési-kilépési eseményeinél is szükségünk lehet a stack-ek megfigyelhetőségére. A stackwalkhoz a `PROC_THREAD` és `LOADER` flageket mindenképp be kell kapcsolnunk. A „stackwalkolható” események listáját megkaphatjuk a következő hívással:

```
xperf –help stackwalk
```

Egy példa-hívás:

```
xperf –on PROC_THREAD+LOADER+SYSCALL –stackwalk SysCallEnter+SysCallExit
```

Megjegyzendő, hogy a bár a stackek rögzítésre kerülnek, a szimbólumok (pl. függvények nevei) nem. Így a megjelenítőeszközben a szimbólumokat megfelelő forrásból be kell majd tölteni, és fel kell majd oldani később.

5.1.4 Leállítás és mentés

Egy futó munkamenet leállítását és egyúttal a leállított munkamenetek egy `.etl` kiterjesztésű állományba mentését a `stop` és `d` kapcsolók segítségével végezhetjük:

```
xperf –stop [<munkamenet_nev>] –d <output.etl>
```

5.2 ETW megjelenítés

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

A 10. ábra a WPR felhasználói felületének alapvető elemeit szemlélteti. A grafikon böngészőből „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 11. 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).

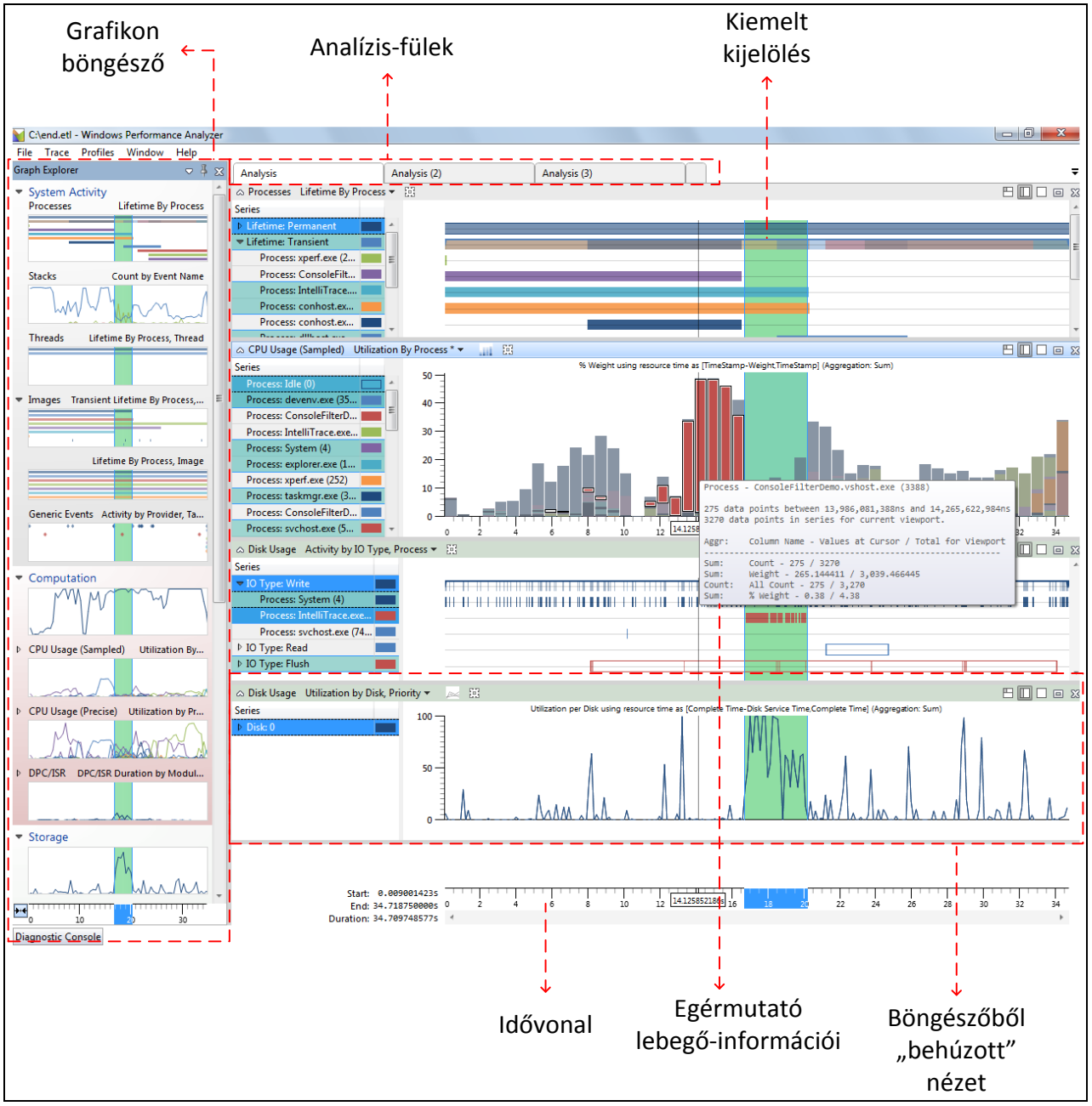
- Először azonosítjuk, hogy ezek a műveletek írások és az `IntelliTrace.exe` végzi őket (harmadik nézet, „Disk Usage – Activity by IO Type, Process”.).
- Észleljük, hogy a gyakori írásokat jelentő szakasz egybeesik az `IntelliTrace.exe` futásának végével (legelső nézet).

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 közpen 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 12. 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 platformnyomkövetővel.



10. ábra. Windows Performance Analyzer.

11. 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

12. 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.)

6 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. http://dsecrg.com/pages/pub/show.php?id=43

[4] MSDN. How to: Use a Symbol Server. http://msdn.microsoft.com/en-us/library/b8ttk8zy.aspx

[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