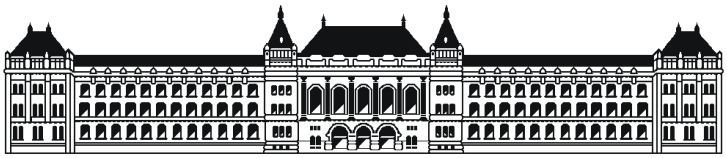


Budapesti Műszaki és Gazdaságtudományi Egyetem
Méréstechnika és Információs Rendszerek Tanszék

Memória profiling lehetőségeinek vizsgálata

Diplomaterv

Készítette
Szabó István Ákos

Konzulens
Lazányi János

2012. december 4.

DIPLOMATERV-FELADAT

Szabó István Ákos (DLBPMP)
szigorló villamosmérnök hallgató részére

Memória profiling lehetőségeinek vizsgálata

A több processzor magot is tartalmazó beágyazott vezérlő áramkörök életünk részévé kezdenek válni. Napjainkban nem ritka, hogy egy mobiltelefonban is négymagos processzort alkalmaznak. Ezen készülékek valamilyen beágyazott operációs rendszert futtatnak, és a több processzor mag hatékonyan kihasználható, a kernel ütemező által.

Olyan stream jellegű alkalmazásoknál azonban, ahol csupán egyetlen konkrét feladatot (pl. hang-, kép- vagy videó-tömörítés) kell nagymennyiségű adaton végrehajtani, a fenti architektúrák megkívánják a program partícionálását. Ezen probléma korunk egyik legnagyobb szoftveres kihívása, hiszen a C nyelven írt programok döntő többsége nincs felkészítve a párhuzamos erőforrások kihasználására.

A feladat párhuzamosíthatóságát sok módszerrel vizsgálhatjuk. A hallgató feladata megvizsgálni egy újfajta megközelítés használhatóságát. Egyik ilyen lehetőség az egyes funkcionális blokkok közötti kommunikáció monitorozása, hiszen az egymással keveset kommunikáló kódrészletek nagy valószínűséggel jól szeparálhatóak.

A hallgató feladatának a következőkre kell kiterjednie:

- Dolgozza fel a témában elérhető szakirodalmat!
- Dolgozzon ki eljárást egy program függvényei közötti kommunikáció monitorozására!
- Ábrázolja a profiling eredményeket grafikusan!
- Értékelje az elért eredményt, alkosson véleményt a memória-profiling ilyen irányú alkalmazhatóságáról!

Tanszéki konzulens: Lazányi János, tanársegéd

Budapest, 2011. március 9.

.....
Dr. Jobbágy Ákos
tanszékvezető

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

1117 Budapest, Magyar Tudósok krt. 2. I. ép. I.E.444.
Telefon: 463-2057, Fax: 463-4112
http://www.mit.bme.hu • e-mail: mitadm@mit.bme.hu

HALLGATÓI NYILATKOZAT

Alulírott *Szabó István Ákos*, a Budapesti Műszaki és Gazdaságtudományi Egyetem hallgatója kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, és a szakdolgozatban csak a megadott forrásokat használtam fel. Minden olyan részt, amelyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

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

Budapest, 2012. december 4.

Szabó István Ákos
hallgató

Tartalomjegyzék

Kivonat

Ahogy napjainkban a többprocesszoros rendszerek egyre szélesebb körben terjednek, fontos kérdés lehet, hogy az ezeken a rendszereken futtatott algoritmusok milyen mértékben képesek kihasználni a több processzor által biztosított erőforrásokat. A használt algoritmusok között találhatóak olyanok, amelyek jól párhuzamosíthatóak, azonban néhány – jellemzően a stream jellegű (videó, hang) feldolgozó – algoritmus esetén ez a feladat igen nehezen kivitelezhető. Az algoritmusokban meglévő párhuzamosítási lehetőségek megtalálására szolgáló automatizált eszköz elkészítése igen komplex feladat, azonban nagymértékben befolyásolhatja az algoritmusok tervezését és megvalósítását.

Az algoritmusok párhuzamosításának első lépése azok mélyreható vizsgálata. A dolgozatomban egy az algoritmusok párhuzamosíthatósági szempontból történő vizsgálatát elősegítő eljárást és az azt megvalósító eszköz elkészítését mutatom be.

Az elérhető szakirodalom feldolgozása által megismert analízis módszerekből kiindulva, a dinamikus adatfolyam gráfok alapján, egy új vizsgálati módszert hoztam létre, amelynek lényege az adatfolyam gráfok aggregálása. A vizsgálat elvégzéséhez létrehozott eszköz a vizsgált program utasításainak futás közbeni követésével és a kinyert adatok feldolgozásával állítja elő az aggregált adatfolyam gráfokat.

Az elkészített analízis eszközt egy JPEG kitömörítő algoritmus vizsgálatával próbáltam ki. Ennek során az aggregált adatfolyam gráf felvétele igen jól használható eszköznek bizonyult, amely nemcsak a program részeinek kapcsolatait és azok működését jeleníti meg nagyon jól használható módon, de a párhuzamosíthatóság vizsgálatának szempontjából is hasznos lehet.

A jelen diplomatervben bemutatott analízis eszköz tehát több szempontból is igen hatékony, és nagy valószínűséggel az automatizált párhuzamosítás megvalósításához is jó kiindulási alap lehet, azonban egy ilyen eszköz elkészítéséhez még jelentős további kutatások elvégzése szükséges.

Abstract

In these days the number of multiprocessor systems gradually increase. Therefore it is an important question whether the algorithms running on these systems are using the resources efficiently. Many algorithms can be easily modified to run on multiple processors, but some (mostly stream-based ones for video and audio processing) can make this task very difficult. Creating a tool that could solve this problem automatically is a very complex task, but it would greatly change the design and implementation of these algorithms.

The first step of modifying the algorithms is an in depth analysis. In this thesis I propose a method that could be used in this step. The implementation of an analysis tool is also discussed.

The studied literature describes some analysis methods, including the dynamic dataflow graphs. Based on this method I created a new, more usable analysis process, that aggregates the dataflow graphs to extract their essence.

I tested the developed tool by analyzing a JPEG decoder algorithm. During this analysis I found, that the aggregated dataflow graph is a very powerful tool for discovering the internal connections of a given algorithm.

In conclusion the analysis tool described in this thesis is very effective and usable, and might serve as a base for developing the aforementioned automatic tool.

Bevezető

Napjainkban egyre több – akár beágyazott – rendszerben találhatunk többmagos processzorokat. Ezekben a rendszerekben a több processzormag biztosította erőforrások megfelelő kihasználása fontos kérdéseket vethet fel. Ilyen kérdés lehet, hogy az eredendően nem párhuzamos jellegű feladatokat hogyan oszthatjuk fel a több processzormag között, hogy azok kihasználtsága optimális legyen.

Azokban az esetekben amikor a rendszerben több egymástól független feladat hajtódik végre, az egyes feladatok végrehajtása könnyedén szétoszthető, hiszen nincs szükség közvetlen együttműködésre közöttük. Amikor azonban csak egyetlen nagyobb feladat megoldása a cél, a részfeladatokra osztás problémákba ütközhet. Bizonyos algoritmusok esetén a felosztás természetesen itt is megoldható, de például a stream jellegű (videó, kép, hang) alkalmazásokra ez nem igaz. Itt a problémát az jelenti, hogy a feldolgozáshoz általában nem elegendő a bemenet egy kis részét ismerni, illetve a feladat nem osztható fel jól egymást követő és egymástól elkülöníthető lépésekre.

Az algoritmusokban meglévő párhuzamosítási lehetőségek megtalálásához azok mélyreható vizsgálata az első lépés. Ez történhet az algoritmus működésének megértése és feltérképezése által, azonban célravezetőbb lehet valamilyen analízis eszköz használata. Többféle ilyen elemző eszköz érhető el, a hívási gráfoktól a dinamikus függőségi gráfokig, amelyek segítséget nyújthatnak az algoritmusok párhuzamosítását illetően. A dolgozatomban ezen eszközök alapján és ezekre építve egy új analízis módszert alkottam meg, amely igen jól használható az algoritmusok párhuzamosíthatóság szempontjából történő vizsgálatához, de egyéb alkalmazások is elképzelhetők.

Az egész éves munka során az elérhető szakirodalmakat feldolgozva megismertem a különféle profiling lehetőségeket és az ezeket megvalósító eszközöket. A statikus és dinamikus hívási gráfok, dinamikus adatfolyam és függőségi gráfok mind jól használhatók programok és algoritmusok különféle szempontok szerinti elemzéséhez. A szakirodalom további tanulmányozása során a programok utasításainak különböző szinteken történő párhuzamosítását korlátozó függőségeket és egyéb tényezőket is áttekintettem. A profiling lehetőségeket az 1. fejezetben mutatom be, míg a párhuzamosítás problémáit a 2. fejezet tárgyalja.

Az aggregált adatfolyam gráfok felvételének első lépése a vizsgált program futási információinak kinyerése. Ezen információk alapján az elkészített analízis eszköz előállítja a program hívási gráfját, dinamikus adatfolyam gráfját, valamint aggregált adatfolyam gráfját is. Az aggregált adatfolyam egyszerű megjelenítése mellett, annak egyéb nézetei is előállíthatók. A 4. fejezetben a vizsgálatok elvégzéséhez használt környezetet és eszközö-

ket mutatom be, azaz például az alkalmazott processzor szimulátort és a gráf kimenetek megjelenítésére használt eszközt. Az elkészített analízis eszköz leírását az 5. fejezet tartalmazza. A követelmények felsorolása után a magas szintű felépítés bemutatása következik. A működés részletes leírása is megtalálható ebben a fejezetben.

Az aggregálás módszerét és az elkészített analízis eszközt egy valódi program esetében is kipróbáltam. Ennek során a módszer használhatóságát vizsgáltam több szempontból is. A vizsgált JPEG dekódoló algoritmus szerkezetének feltárása közben az aggregált adatfolyam gráfok igen hasznosnak és jól használhatónak bizonyultak. A megszerzett tapasztalatokat a 6. fejezetben írtam le. A párhuzamosíthatóság vizsgálatára vonatkozó néhány megjegyzés is itt olvasható.

A 7. fejezetben végül összefoglalom az elvégzett munkát, és bemutatom a lehetséges továbbfejlesztési irányokat.

1. fejezet

A feladat előzményei

Ebben a fejezetben – a tématerület szakirodalmának feldolgozása alapján – a megoldandó feladat előzményeit és a profiling területén elérhető megoldásokat mutatom be. Az általános profiling fogalmának ismertetése után rátérek a memória profiling speciális megoldásainak áttekintésére, kiemelve az elérhető szoftvereszközök lehetőségeit és hátrányait.

1.1. Profiling általánosan

Mielőtt a hagyományos memória profiling bemutatására rátérnék, fontosnak érzem bemutatni a tágabban értelmezett profiling-ot is, amely a szoftverfejlesztés egyik fontos eszköze. A profiling[1] során általában az adott program egyes részeinek, függvényeinek futási idejét szeretnénk meghatározni, hogy a kritikus (sokszor futó) részek azonosítása után, azok implementációját optimalizálva, a teljes algoritmus működését gyorsabbá tudjuk tenni.

A profiling kimenete általában a program adott futására vonatkozóan megadja, hogy az egyes utasítások, illetve függvények hány alkalommal futottak le. Gyakran a futások számán túl az egyes lépések elvégzéséhez szükséges időt is megkaphatjuk.

A profiling célja többféle lehet, a szoftver aktuális fejlesztési állapotától függően. Az implementáció közben például az algoritmusok megvalósításának optimalizálásához a futási idő információk használhatók. A tesztelés során az egyes utasítások lefutásának számát meghatározva, a tesztelés hatékonyságára vonatkozó tesztfedettségi mérőszámok határozhatók meg.

1.1.1. Hívási gráfok

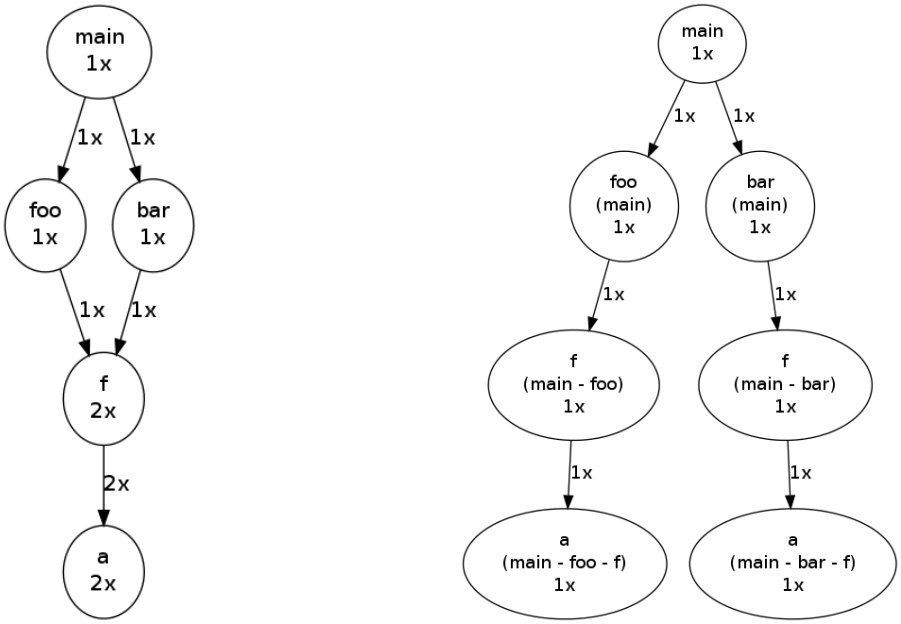
A profiler-ek kimeneteként gyakran a vizsgált program hívási fája is megkapható[1]. A hívási fa lényegében egy gráf, amelyen a pontok reprezentálják a program egyes függvényeit, míg az irányított élek mutatják a köztük lévő függvényhívásokat. Azaz egy pontból akkor mutat él egy másikba, ha az egyik függvény meghívta a másikat. A hívási fák előállításuk szerint két csoportba sorolhatók. Ez a két csoport a statikus és dinamikus fák.

A statikus hívási fák előállítása a program statikus analízise alapján rajzolható fel, azaz a program tényleges futtatása nem szükséges az előállításához. Előnye, hogy tartalmazza az összes függvényt és az összes (nem függvénypointeren keresztüli) lehetséges függvényhívást,

azonban a program futtatására vonatkozó információkkal nem szolgál. A statikus hívási fákban, létrehozásukból adódóan olyan hívási utak is szerepelhetnek, amelyek a program tényleges futtatása során nem következhetnek be.

A dinamikus hívási fák a vizsgált program egy adott futtatására vonatkozó függvényhívási kapcsolatokat mutatják, azaz a felvételéhez a programot a futása közben kell analizálni. A dinamikus hívási gráf előnye, hogy tartalmazza az adott lefutáshoz tartozó összes függvényhívást (még a függvénypointereken keresztüli hívásokat is), azonban csak ezeket tartalmazza, azaz például más bemenő paraméterek esetén kaphatunk lényegesen eltérő gráfot is.

A hívási gráfokat egy másik szempont alapján is csoportosíthatjuk aszerint, hogy az egyes függvényekre vonatkozóan megkülönbözteti-e a függvény hívásának forrását. Így megkülönböztethetünk kontextus érzékeny (context-sensitive) és érzéketlen (context-insensitive) hívási fákat. A kontextus érzéketlen hívási gráfok esetén egy függvényhez minden esetben egyetlen gráf csomópont fog tartozni, míg a kontextus-érzékeny esetben egy függvényt több csomópont is reprezentálhat. Ezek között a pontok között a különbséget az eltérő hívási utak jelentik.



(a) *Kontextus érzéketlen eset* **(b)** *Kontextus érzékeny eset*

1.1. ábra. *Hívási gráf példák*

Az 1.1. ábrán a kontextus érzékeny és érzéketlen hívási gráfok közötti különbség látható egy példán. A hívásokat jelentő gráf-éleken a hívások száma látható. A kontextus érzéketlen esetben az „a" függvényről csupán annyit tudunk, hogy az „f" függvény két alkalommal hívta meg. Ez az eredmény azonban többféle módon is létrejöhet. Lehetséges, hogy „f" csak a „foo" vagy a „bar" függvényből történő meghívásra használja „a"-t, de az is, hogy mindkét esetben meghívja. Ezt a bizonytalanságot szünteti meg a kontextus érzékeny megjelenítés. Ahogyan az ábrán is látható, ekkor minden függvényhez több csomópont is tartozhat a

hívási kontextus szerint. Ezen ábra alapján már pontosan meg tudjuk mondani, hogy „f” mindkét esetben egyszer hívja meg az „a” függvényt.

A kétféle megjelenítési forma közül egyértelműen a kontextus érzékeny használata ad több információt a program működéséről, viszont ezzel összhangban ennek mérete és például az előállításának memóriaigénye jelentősen nagyobb, ezért bizonyos program méret fölött már csak korlátozásokkal alkalmazható.

Ahogy az 1.1. ábrán is látható a hívási fa és az egyes függvények profiling információi (futások száma) jól megjeleníthetőek egy közös ábrán. Ezt a megoldást gyakran alkalmazzák a profiler alkalmazások is.

1.1.2. Profiling adatok gyűjtése

A profiler-ek működéséhez a vizsgált program futásáról információkat kell gyűjteni, azaz valahogyan nyomon kell követni a végrehajtást. Ennek megvalósítása többféle módon lehetséges[2]. Az egyik lehetséges eljárás a futtatott kód „felműszerezése”, azaz a program forráskódjának kiegészítése speciális utasításokkal, amelyek adatokat szolgáltatnak a program működéséről. Ezzel a megoldással a program futása jól követhető, azonban az extra utasítások lényeges sebességcsökkenést eredményeznek, vagy akár a program hibás működését is okozhatják. A felműszerezés megvalósítható kézzel, de például a fordító közreműködésével is. A manuális felműszerezésre példa lehet a beágyazott rendszerek esetén gyakran alkalmazott GPIO kimenet „billegtetés”, amikor a program kritikus pontjaira elhelyezett utasítások néhány GPIO kimenet állapotát változtatják, és így a lefutás a kimenetek alapján követhető.

A fordító általi felműszerezésre egy példa a gcc fordító `-pg` kapcsolója. Ekkor a fordító a program utasításai közé helyezi el a profiling adatokat szolgáltató utasításokat, majd ezek a program futása közben az összegyűjtött információkat egy fájlba írják. Ezt a fájlt a továbbiakban bemutatásra kerülő gprof (l. 1.3.1) elemzi, és meghatározza a program hívási gráfját, illetve az egyes függvényekben eltöltött időt. Természetesen ez az eljárás sebességcsökkenést eredményezhet, illetve a program újrafordítása szükséges a felműszerezéshez.

A profiling-hoz szükséges adatok gyűjtésének egy másik módja, ha a vizsgált program bináris kódját dinamikusan, futási időben átfordítjuk egy egyszerű, processzor független formába, amit ezután egy szimulált, virtuális processzoron futtatunk. Ekkor a szükséges futási adatok a közbenső formában lévő kódból nyerhetők ki. Ezt az eljárást alkalmazza a később (l. 1.3.2) bemutatásra kerülő Valgrind is. Mivel ebben az esetben a profiling folyamat bemenete a lefordított bináris, nincs szükség a forrás újrafordítására, illetve magára a forráskódra sem. A megoldás hátránya, hogy a futási időben történő extra műveletek jelentős időt igényelnek, azaz a program futása lényegesen lassabb lesz.

Amíg az eddig bemutatott mindkét adatgyűjtési eljárás során a program futása valamilyen módon megváltoztatásra került, enélkül is lehetséges a program megfigyelése. Az előzőekben bemutatott eljárások közös pontja volt, hogy a futó program megfigyelését szoftveresen (felülről) végezték. A következő két eljárás ezzel szemben a profiling-hoz szükséges adatokat közvetlenül a gépi kódú utasítások végrehajtása alapján gyűjti.

Az egyik lehetőség az ilyen jellegű adatgyűjtésre, a processzor-, pontosabban utasításkészlet szimulátor használata. Ezek a szoftverek a processzor működését szimulálják utasításonként. A szimulált processzor regiszterei tulajdonképpen változók, amelyeken az utasításoknak megfelelő műveleteket hajtja végre a szimulátor szoftver. A processzor környezete (memória, perifériák) hasonló módon szimulálható.

Látható, hogy a felépítésből adódóan a futó program részletes (utasítás és regiszter szintű) megfigyelése nagyon könnyen megvalósítható ezzel a megoldással. További előny, hogy mivel a szimulált processzoron futó kód nem kerül végrehajtásra a valódi processzoron, lényegében bármilyen architektúra szimulálható.

A második lehetőség a gépi kódú utasítások alapján történő információgyűjtésnek a tényleges hardver használata. Ekkor természetesen valamilyen módon követni kell a valós processzor működését. Ennek megvalósíthatósága erősen függ a vizsgált processzortól és a környezettől. Ha a processzort FPGA-ban implementáltuk, a szükséges belső cím- és adatvezetékek kivezetése és megfigyelése viszonylag könnyű feladat, akár a belső regiszterek is megfigyelhetők ilyen módon. Speciális eset, ha a processzor(mag) kifejezetten a program futásának követését lehetővé tévő egységeket is tartalmaz. Ilyen egységeket találhatunk az ARM újabb generációs processzormagjaiban (CoreSight). Ezek az egységek lehetővé teszik a processzoron futó program valós környezetben történő megfigyelését, a futás befolyásolása nélkül.

A bemutatott lehetőségek csak egy részhalmazát alkotják a profiling információk kinyerését lehetővé tévő megoldásoknak, azonban a továbbiakat itt nem ismertetem.

1.2. Hagyományos memória profiling

A hagyományos memória profiling[3] használatával a program memóriakezelésével kapcsolatos hibákat és problémákat találhatjuk meg. Sok ilyen hiba a dinamikus memóriák használatából ered. A dinamikus memóriakezelés a bonyolultabb programok esetében szinte elkerülhetetlen, azonban sok veszélyt hordozhat magában a használata, ilyenek például a nem lefoglalt, vagy már felszabadított memóriaterület használata, illetve egy terület többszöri vagy egyszer sem elvégzett felszabadítása. További hibák léphetnek fel pointerek és tömbök használatakor, például egy adott méretű tömb túlindexelése.

Egyértelműen programozási hiba, amennyiben egy memóriaterületet (például lokális változót, heap területet) inicializálás nélkül használunk fel, azonban az ilyen hibák nem mindig okoznak észrevehető hibajelenséget, illetve a hiba okának megtalálása is nehézkes lehet. Ilyen esetekben is célszerű lehet memória profiler használata, amelyek a felsorolt hibák legnagyobb részét feltárják, és ezzel lehetőséget adnak a hibák kijavítására.

1.3. Profiling alkalmazások

A következőkben a gprof és a Valgrind példáján keresztül mutatok be két eltérő működésű profiling rendszert.

1.3.1. GNU gprof

A nyílt forráskódú gprof egy jól használható profiling szoftver, amely alkalmas nagy, bonyolult programok vizsgálatához is [1]. Mint a profiling megoldások legnagyobb része, a gprof is képes meghatározni egy adott program futtatása során, hogy a program függvényei közül melyik hány alkalommal futott, illetve a futása mennyi időt vett igénybe. Megkapható továbbá a program hívási gráfja is, amely tartalmazza, hogy az egyes függvények milyen függvényeket hívtak a futásuk során. A profiling célja ebben az esetben is a program jobb megértése, illetve az optimalizálandó részek megtalálásának segítése.

A felműszerezés szempontjából a gprof a fordító által segített felműszerezést alkalmazza, azaz a vizsgált program újrafordítása szükséges a profiling adatok gyűjtésének lehetővé tételéhez. A felműszerezést a GNU gcc fordító `-pg` kapcsolójával engedélyezhetjük a program fordításánál. Az így fordított programok esetén a gcc minden egyes függvény prológusába beilleszt egy extra függvényhívást egy monitorozó függvényre. Ez a függvény végzi lényegében a profiling feladatok nagy részét.

A monitorozó függvény minden meghívásakor a stack-en található visszatérési cím alapján meghatározza, hogy melyik függvényből hívták meg, azaz a program futása során melyik függvényre került a vezérlés. Ezután hasonló módon meghatározza, hogy honnan érkezett az előző függvényhívás. Így megkapható, hogy az adott függvényhívás honnan érkezett (hívó) és melyik függvényt aktiválta (hívott). A monitorozó függvény ezzel tulajdonképpen a hívási gráf egy élét kapja meg. Ha a profiling során az adott él már szerepelt, akkor az azon található hívási számláló növelésével jegyezhető fel a hívás ténye, ellenkező esetben pedig egy új él létrehozása szükséges.

Az egyes függvények futási idejének meghatározására két lehetőség adódik. Ha a függvények futásához szükséges időt teljesen pontosan szeretnénk meghatározni, szükség van a függvénybe belépés és a kilépés közötti idő mérésére. Ez a megoldás első ránézésre megvalósítható, azonban mivel az operációs rendszerek a több feladat párhuzamos futtatását időosztásosan oldják meg, a belépés és a kilépés között eltelt idő az adott függvény futtatásához szükséges időről lényegében nem ad semmilyen információt. Ahhoz, hogy a pontos idő mérése lehetséges legyen, az operációs rendszer ütemezőjének közreműködésére is szükség lehet, hogy az egyes feladatok közötti átkapcsolásokról is értesülhessünk. Mivel ez a megoldás bonyolult és időigényes lehet, a statisztikai alapú becslés hatékonyabb módszer.

Ha egy program futás során megfelelő gyakorisággal elmentjük a programszámláló értékét, majd ez alapján meghatározzuk, hogy a program éppen melyik függvényt hajtotta végre, képet kaphatunk az egyes függvényekben eltöltött idő mértékéről. A program azokban a függvényekben tölt sok időt, amelyek hosszúak vagy sokszor futnak. Így tehát a program teljes futási idejéből, az egyes függvények futásainak számából és a rögzített programszámláló értékekből becslés adható a függvények futási idejeire. A kapott idők a mérés statisztikus jellegéből adódóan nem lesznek teljesen pontosak.

A gprof is ezt az eljárást használja a futási idők meghatározásához. Egy megfelelő gyakoriságú megszakításban hisztogramot készít a programszámláló állásáról, és ezt felhasználva határozza meg a függvények közelítő futási idejét.

A gprof tehát a fordító segítségével helyezi el a profiling adatok gyűjtéséhez szükséges függvényhívásokat a monitorozott programban. A vizsgált program futtatásakor elsőként a profiling adatokat tároló struktúrák felépítése történik meg, majd ezután kerül a vezérlés magára a programra. A program futása közben ezekbe a struktúrákba írja a monitorozó függvény az adatokat, majd a futás végén történik meg az adatok előfeldolgozása és mentése.

Miután a program futása befejeződött, a gyűjtött adatokat tartalmazó fájlt át lehet adni magának a gprof szoftvernek, ami elvégzi az adatok feldolgozását, és megjeleníti a profiling eredményeit. Ekkor történik meg a hívási gráf és a futási idő információk összefésülése, azaz annak meghatározása hogy egy függvény futása mennyi ideig tartott önmagában, illetve mennyi a futási idő az általa hívott más függvények futási idejével együtt.

A gprof a profiling eredményeket karakteres módon, táblázatosan jeleníti meg, azonban a további felhasználást megkönnyítendő, elérhetők grafikus megjelenítők is, mint például a Kprof[4] vagy az Xgprof[5].

1.3.2. Valgrind

A Valgrind[6] egy nyílt forráskódú DBI (Dynamic Binary Instrumentation – dinamikus bináris felműszerező) keretrendszer, amellyel nagy bonyolultságú DBA (Dynamic Binary Analysis – dinamikus bináris analízis) eszközök készíthetők. Az elkészíthető eszközök képességei jelentősen meghaladhatják az egyszerű profiling alkalmazások képességeit. A DBI eszközök a vizsgált program felműszerezését annak újrafordítása, vagy akár a forráskód ismerete nélkül végzik, futási időben. A vizsgálathoz szükséges utasítások a dinamikus újrafordítási technika használatával kerülnek be a végrehajtandó kódba, annak futtatása előtt. A DBI eszközök lényeges előnye, hogy a teljes (az operációs rendszer kernel hívásait kivéve) futtatott kód felműszerezése elvégezhető.

A Valgrind egyik érdekes és kiemelkedően megvalósított tulajdonsága, hogy lehetőséget ad az úgynevezett árnyékértékek használatára. Az árnyékértékek lényege, hogy minden regiszterhez és memóriarekeszhez tartozik egy ilyen változó, amelyben a tényleges regiszterben vagy memóriacímen lévő adatról tárolhatunk valamilyen többletinformációt, metaadatot. Ez a lehetőség sokféle nagy tudású DBA eszköz létrehozásához ad segítséget. Egy egyszerűbb esetben az árnyékértékek például tárolhatják, hogy az adott memóriacím vagy regiszter inicializált értéket tartalmaz-e. Így egy program futtatása során egyszerűen megtalálhatóak a nem inicializált változóhasználatok, amelyek akár nem is minden esetben okoznak hibás működést.

A Valgrind keretrendszerhez sokféle DBA eszköz érhető el, amelyekkel programok működésének különböző aspektusai vizsgálhatók.

A vizsgálatok elvégzéséhez tehát szükség van a Valgrind keretrendszerre, amely elvégzi a program felműszerezéséhez szükséges (később bemutatásra kerülő) lépéseket, illetve egy DBA eszközre, amely feladata a ténylegesen beszúrandó extra műveletek megadása, és a kapott eredmények feldolgozása. Ennek a felépítésnek az az előnye, hogy amíg a keretrendszer igen komplex, az egyes DBA eszközök felépítése és megvalósítása a feladattól függően

lehet viszonylag egyszerű is. A különböző feladatokhoz is jól illeszkedő keretrendszer használatával nincs szükség minden egyes eszközhöz saját keretrendszerre, ami jelentősen felgyorsíthatja a fejlesztést.

A következő részben a Valgrind és a hozzá illeszkedő eszközök működését mutatom be részletesen.

Működése

A Valgrind keretrendszer működésének egyik legfontosabb részlete a dinamikus bináris újrafordítás, amellyel a vizsgált programok felműszerezése és futtatása elvégezhető. Ennek lényege, hogy a futtatandó kódot a rendszer gépi utasításonként átfordítja egy köztes reprezentációra (IR – Intermediate Representation), majd ehhez adja hozzá a DBA eszköz a felműszerezéshez szükséges, szintén IR nyelvű utasításait. Ezután a köztes reprezentációról a rendszer elvégzi a visszafordítást a rendszer gépi kódjára, majd az eredményt lefuttatja. Fontos, hogy a vizsgált program nem önálló folyamatként fut, annak végrehajtását minden szempontból a keretrendszer felügyeli.

A vizsgált program fordítása és felműszerezése úgynevezett blokkonként történik. A blokkok feldolgozása után az eredmény gyorsítótárba kerül, hogy az ismételt futtatás esetén ne kelljen a fordítás időigényes feladatát ismét elvégezni. A blokkok feldolgozása (fordítás, felműszerezés) a program futásának megfelelően hajtódik végre, azaz mindig csak az aktuálisan végrehajtásra kerülő részek kerülnek feldolgozásra. Megfigyelhető, hogy a működési módból adódóan az eredeti vizsgált kód egyetlen utasítása sem kerül közvetlenül végrehajtásra.

A Valgrind a dinamikus bináris újrafordítás alkalmazása folytán azokat a külső programkönyvtárakat (dinamikusan linkelt, osztott) is képes felműszerezni és megfigyelni, amelyeket más elven működő profiling eszközök nem. Ez bizonyos helyzetekben lényeges előny lehet.

A Valgrind indításához a felhasználó a vizsgálni kívánt program és a használandó DBA eszköz nevét adja meg paraméterként. Ennek hatására elindul a keretrendszer, amely elsőként elvégzi a futtatáshoz szükséges alrendszerek betöltését. Ezután a keretrendszer betölti a futtatandó programot, majd következő lépésként inicializálja a kiválasztott DBA eszközt. Ekkor történik meg a parancssorban megadott beállítások feldolgozása is. Miután az összes szükséges alrendszer működésre kész állapotba került, a keretrendszer megkezdi a vizsgált kód átfordítását és futtatását a megfelelő címtől kezdve.

Hogy a Valgrind a vizsgálandó program bináris állományát a keretrendszer folyamatába tölti be, lényeges előnyt jelenthet. Egy eltérő megközelítés lehet, ha – fordított módon – a DBI rendszert kapcsoljuk a már elindított analizálni kívánt programhoz. Ez azonban több problémát is okozhat, hiszen ekkor a vizsgált programnak lehetnek olyan részei, amelyek még a DBI rendszer betöltése előtt lefutnak és így nem vizsgálhatók. Ezen túl, mivel a betöltéshez az operációs rendszer közreműködésére is szükség van, ez a megoldás túlságosan függ a rendszerkörnyezettől is.

Felműszerezés

A DBA keretrendszerek fő feladata a bináris formában rendelkezésre álló program felműszerezése, majd futtatása. Ennek megvalósítására alapvetően két különböző megoldás használható.

Az egyik lehetőség a kód „elemi lépésekre" bontása, a felműszerezés hozzáadása, majd az eredmény újraszintetizálása futtatható kóddá (disassemble and resynthesize). Ennek első lépéseként a keretrendszer a vizsgálandó program gépi utasításait egy köztes reprezentáció (IR) utasításaivá alakítja. Ez a megjelenítés teljesen processzorfüggetlen, azaz az eredeti utasítások egy, de akár több IR utasítássá képződhetnek le. Ekkor a használt DBA eszköz az átfordítás eredményét megkapva, annak utasításai között helyezheti el a saját felműszerező utasításait, amelyek szintén a köztes reprezentáció szerint vannak megadva. A keretrendszer ezután az így módosított kódot visszafordíthatja az adott architektúra gépi utasításaira, majd az eredmény lefuttatható.

Az IR utasításokra való fordítás lényege, hogy azok az eredeti utasítások minden hatását tartalmazzák, így a későbbi visszafordítás után a futtatás eredménye meg fog egyezni az eredeti kód által adott eredménnyel, annak ellenére, hogy az eredeti kód egyetlen utasítása sem kerül közvetlenül végrehajtásra.

Ez a megközelítés, ahogyan a további lehetőségek bemutatásánál látható, jelentősen bonyolultabb lehet, azonban előnyei miatt mégis jól használható. A megvalósítás szempontjából a disassemble and resynthesize eljárás viszonylag bonyolult, az átfordítások utáni optimalizálásokhoz a fordítóprogramok esetén használt algoritmusokat és technológiákat is igénybe kell venni. Ettől a fordítás és felműszerezés jelentős mennyiségű időt vehet igénybe, ami a program futását nagymértékben lelassítja. További hátrány, hogy az ismét gépi kódra visszafordított utasítások az eredeti utasításokról alapvetően nem adnak információt, így amennyiben az analízis célja a program utasításainak alacsony szintű vizsgálata, további lépésekre van szükség.

A disassemble and resynthesize elven működő DBI eszközök lényeges előnye a nagy komplexitású analízis eszközök megvalósításában van. Ezeknél az eszközöknél a vizsgált program kódjába nagy mennyiségű és bonyolult kiegészítő utasításokat kell beszúrni, amelynek megvalósítása a köztes reprezentációt alkalmazva jóval hatékonyabb lehet, illetve a fordításokkor alkalmazott optimalizálási lépések miatt a végső teljesítmény is jobb lehet a más eljárásoknál tapasztalhatónál.

Egy ettől eltérő megközelítés a copy and annotate (másolás és felcímkézés). Ennek lényege, hogy a vizsgált program utasításai lényegében módosítás nélkül kerülnek végrehajtásra, azonban a keretrendszer minden utasításhoz feljegyzi annak leírását, azaz a különféle hatásait. A DBA eszközök ezeket a leírásokat felhasználva állíthatják elő a felműszerező utasításokat, amelyeket ezután a keretrendszer fésül össze az eredeti utasításokkal.

Ennek a megoldásnak lényegében az utasítások összefésülése a legkényesebb pontja. Ügyelni kell a folyamat során arra, hogy a beszúrt utasítások ne zavarják meg az eredeti program működését, ne okozzanak hibás működést. A felműszerezéshez egyszerűbb esetben utasításokat, komplexebb funkcionalitás esetén függvényhívásokat kell általában beszúrni.

A két forrásból érkező utasítások összefésülése az eddigieken túl, teljesítménybeli problémákat is okozhat, hiszen amint látható, ennél a megoldásnál hiányoznak a disassemble and resynthesize esetén alkalmazott optimalizálási lépések.

A bemutatott két felműszerezési megoldás nem csak külön-külön, de egymás mellett is használható (hibrid megoldás). Ekkor a rendszer bizonyos utasítások esetén az egyik, míg más utasítások esetén a másik megoldást használják, azok előnyeit összekapcsolva. Ezt a megoldást alkalmazták a tárgyalt Valgrind korai verziói is.

A Valgrind jelenlegi megvalósítása szerint a disassemble and resynthesize eljárást alkalmazza, mivel ahogyan láthattuk, ez lényegesen jobban illeszkedik a nagy komplexitású DBA eszközök megvalósításához.

Fontos megjegyezni, hogy a vizsgált program kódjának egyes részein a Valgrind általában csak egyszer végzi el a fordítási és felműszerezési lépéseket, a további futtatások alkalmával már csak a korábban felműszerezett és gépi kódra visszafordított utasítások végrehajtása történik.

További érdekesség, hogy a közbenső reprezentáció utasításait nem csak az eredeti gépi kód architektúrájára fordíthatjuk vissza, de elvileg bármilyen architektúrára. Ezzel megvalósítható eltérő processzorra írt programok futtatása. A Valgrind ezt a lehetőséget nem használja ki, azonban hasonló elvek alapján működik a QEMU[7] processzor szimulátor.

A következőkben röviden bemutatom a köztes reprezentációt és a tényleges fordítás és felműszerezés részletes működését.

Dinamikus fordítás

A Valgrind az analizált programok felműszerezését, amint már láthattuk, egy köztes reprezentáció használatával valósítja meg. Ennek a megfelelő megválasztása és kialakítása nagyban befolyásolhatja például az egyes DBA eszközök megvalósítását, illetve a rendszer teljesítményét.

A Valgrind által használt köztes reprezentáció (IR) teljesen processzor független leírási módot ad. Fontos tulajdonsága, hogy az IR utasítások static-single-assignment (SSA) formájúak. Ez azt jelenti, hogy a műveletek leírása során minden átmeneti változó pontosan egyszer kaphat értéket. Egy adott változó egymást követő értékadásai ebben a reprezentációban a változó különböző „változatait” hozzák létre. A Valgrind korlátlan számú átmeneti változót biztosít ezek tárolására.

Az SSA megadási módot alkalmazza sok fordítóprogram is az utasítások belső leírására, mivel nagymértékben megkönnyíti a fordítási és optimalizálási lépéseket. Ahogyan látni fogjuk, a Valgrind is több egymást követő fordítási és optimalizálási lépést alkalmaz, ezért ez a leírás hasonlóan itt is igen előnyös.

Az IR utasításokat a Valgrind utasítás-blokkonként kezeli, ennek részletei a következőkben lesznek láthatóak. Az utasítások mindegyike valamilyen adat tárolását végzi mellékhatásként, ez az SSA forma miatt szükséges. Így adhatunk értéket regisztereknek vagy átmeneti változóknak, illetve tárolhatunk értéket memóriacímre. Az utasítások „paraméterei” kifejezések lehetnek, amelyeknek nincsen mellékhatása, csupán visszatérési értékük.

Ilyen kifejezések lehetnek konstansok, regiszter vagy átmeneti változó olvasások, adat beolvasása adott memóriacímről illetve aritmetikai kifejezések. Példaképpen egy összeadás művelethez a két bemeneti értéket egy-egy kifejezés adja meg.

Az utasításokban szereplő kifejezések megadhatóak egymásba ágyazottan is, fa-szerűen (tree IR). Ekkor a kifejezések bemenő paraméterei is lehetnek kifejezések, tetszőleges mélységig egymásba ágyazva. A feldolgozást nagyban segíti, ha az utasításokban szereplő kifejezések csak egyszeres mélységűek, azaz nincsenek egymásba ágyazva. Ekkor beszélhetünk flat megjelenítésről.

A DBA eszközök megvalósítása igényelheti a program eredeti utasításaira vonatkozó információkat. Amint láttuk, ezek alapvetően nem szerepelnek a köztes reprezentációban, ezért a Valgrind az IR utasítások között elhelyez az eredeti utasításokra utaló „megjegyzéseket", amelyeket az analízis eszközök felhasználhatnak.

A vizsgált program feldolgozásának „nulladik" lépése a kód blokkokra bontása. A Valgrind a blokkok határait néhány egyszerű szabály alapján határozza meg, azonban a feldolgozás így is elég hatékony tud lenni. A szabályokat a keretrendszer a vezérlési utasítások szerint követett programra alkalmazza. Új blokk határát jelenti a feltételes ugrás elérése (hiszen nem ismert a további futás), az ismeretlen címre való ugrás, több mint három feltétel nélküli ugrás ismert címekre, illetve egy bizonyos utasításszám elérése az adott blokkban. Ezen feltételek bármelyikének teljesülésekor a Valgrind megkezdi az adott blokk feldolgozását. A következő felsorolásban ennek lépéseit ismertetem:

Disassembly Első lépésben a keretrendszer a program gépi utasításai alapján létrehozza azok IR leírását. Ekkor minden utasítást egy vagy több IR utasítás fog leírni. A létrejövő IR utasítások tree típusúak, azaz a bennük található kifejezések paraméterei között szerepelhetnek további kifejezések is beágyazva. Az egyes műveletek egymástól függetlenül kerülnek átalakításra, és az összes regisztermővelet expliciten fog szerepelni az eredményként kapott IR leírásban.

Mivel az eredeti utasítások ezen lépés után már nem állnak rendelkezésre, azok közvetett hatásait is le kell írni, így például a processzor státuszregiszterének értékeit is külön IR utasításokkal kell megadni. Az átalakítás után az eredményként kapott reprezentáció, ahogy láthattuk, elég terebélyes és sok redundáns illetve felesleges művelet szerepel benne. Ez a helyzet a következő optimalizálási lépés során jelentősen javulni fog.

Első optimalizálás Ennek a lépésnek alapvetően két feladata van. Az egyik a tree jellegű reprezentációban szereplő beágyazott kifejezések eltávolítása, azaz flat megjelenítéssé alakítás, a másik pedig az előzőekben említett optimalizálás. A flat tulajdonság eléréséhez a beágyazott kifejezéseket átmeneti változók használatával kell alakítani egyszerű kifejezésekké.

Az optimalizáció során eltávolításra kerülnek például a felesleges (redundáns) másolások , illetve egyéb optimalizálási lépések is végrehajtódnak. Ezek között szerepelnek például az elérhetetlen kódrészletek eltávolítása, a konstans változók behelyettesítése,

a közös részkifejezések kezelése és egyéb optimalizálási eljárások. Az erőteljes optimalizáció szükséges, hogy az előző lépésben létrehozott nagymértékű redundanciát is tartalmazó utasítások a felműszerezés során megfelelően kezelhetőek legyenek.

Felműszerezés A vizsgált program felműszerezését nem a keretrendszer végzi, ezt a lépés a DBA eszköz hajtja végre, hiszen a különböző analízis feladatokhoz eltérő extra funkcionalitás szükséges. A flat megjelenítés azért is előnyös ebben a lépésben, mert így nem kell a bonyolult egymásba ágyazott kifejezéseket végigkövetni, sokkal egyszerűbben feldolgozhatóak az utasítások. A felműszerezés során beszúrt utasítások elérhetik a program által használt regisztereket, átmeneti változókat és memóriaterületeket, illetve a korábban ismertetett árnyékértékeket. A felműszerezés tényleges megvalósítására egy példát az 1.3.3. fejezetben mutatok be.

Mivel a DBA eszköz által generált kód szükségszerűen nem illeszkedik a meglévő program folyamába, a keretrendszer a következő lépésben ismételten végrehajt egy optimalizálási lépést a nagyobb hatékonyság érdekében.

Második optimalizálás Ez az optimalizálási lépés a felműszerező utasítások által megváltoztatott kódot dolgozza fel. Itt már elegendő csupán egyszerűbb optimalizálási algoritmusokat végrehajtani, mint például a konstansok behelyettesítése, illetve az elérhetetlen kódrészek eltávolítása.

Tree reprezentáció Ebben a lépésben a felműszerezett és optimalizált kódblokk flat IR utasításait a Valgrind átalakítja egy tree jellegű leírássá. Ennek részeként az átmeneti változók értékadásai behelyettesítésre kerülnek az azokat felhasználó kifejezésekbe, így összetett kifejezéseket létrehozva. Ez a lépés előkészíti a műveleteket leíró gépi utasítások kiválasztását.

Utasítás kiválasztás Miután a köztes reprezentációban lévő utasításokat ismételten gépi kóddá kell alakítani azok futtatása előtt, meg kell találni az egyes IR utasításoknak megfelelő gépi műveleteket. Ezt végzi el ez a lépés.

A feldolgozás eredménye ennél a lépésnél már a processzorfüggő utasítások listája, azonban a regiszterműveleteknél még nem a processzor tényleges regiszterei szerepelnek, hanem csupán virtuális regiszterek, amelyeket a következő lépés fog leképezni a valódi regiszterekre.

Regiszter allokáció Az előző lépésben létrejött utasításlista virtuális regisztereit ez a lépés képzi le a processzor tényleges regisztereire. Ha a rendelkezésre álló regiszterek száma nem elegendő, ebben a lépésben további memóriaműveletek beszúrására lehet szükség, amelyek az éppen nem használt változókat a regiszterek és a memória között mozgatják.

Gépi kód előállítása A fordítás utolsó lépése az előző két lépésben létrehozott utasításlista alapján a tényleges gépi kód előállítása (assembly). Itt lényegében az egyetlen feladat az utasítások és azok paramétereinek megfelelő kódolása, majd az eredmény memóriába írása.

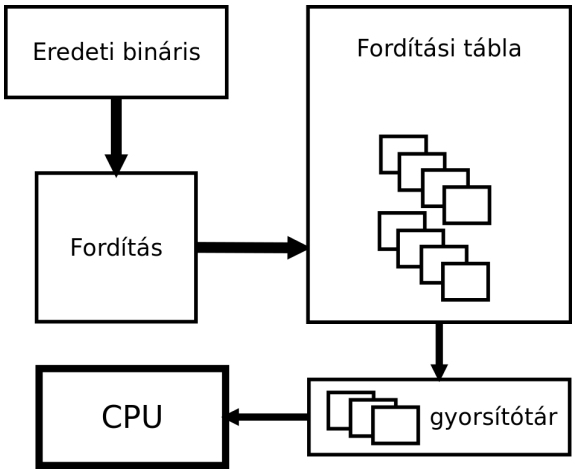
A működési módból adódóan a fordítás során meglepően kevés processzorfüggő lépés van, csupán a gépi kódot közvetlenül kezelő disassembly és assembly, illetve a visszafordításkor az utasítások kiválasztása függ az alkalmazott architektúrától. Ez a tulajdonság elvileg megkönnyítheti a Valgrind különböző architektúrákra való portolását.

A Valgrind a fenti lépések eredményeképpen létrejövő felműszerezett kódokat egy gyorsítótárban tárolja el, ez a fordítási tábla. Ennek a táblának a nagy mérete biztosítja, hogy az analizált program futtatása során egy adott kódblokkot jó esetben elegendő legyen egyetlen alkalommal lefordítani, és így a további futtatások esetén megtakarítható a fordítás jelentős időköltsége. A fordítási tábla megvalósítása fix méretű, ezért szükséges lehet az eltárolt fordítások törlése, amennyiben további kapacitás szükséges.

Futtatás

A felműszerezett kód blokkok futtatásánál fontos szempont, hogy az idő mekkora részében fut a tényleges kód, és mekkora része fordítódik az egyéb feladatokra, hiszen ez nagyban meghatározza a rendszer teljesítményét. A Valgrind esetében ezért a korábban bemutatott fordítási táblán túl még egy kisebb gyorsítótárat is találhatunk.

A kód blokkok futtatása egymás után történik. Minden blokk végén a vezérlés visszakerül egy vezérlőkódra, amely először a kisebb gyorsítótárban keres a következő végrehajtandó kódrészlethez tartozó fordítást. Ha talál ilyet, a vezérlés minimális késleltetéssel kerül tovább a következő kódblokkra. Ha a keresett részlet nem szerepel ebben a gyorsítótárban, a rendszer további keresést végez a fordítási tábla bejegyzései között. Ha itt megtalálható a következő blokk, az bekerül a gyorsítótárba és a vezérlés átadódik rá. Amennyiben itt sincs találat, a rendszernek el kell végeznie az adott blokk fordítását. Ennek eredménye először bekerül a fordítási táblába, majd az előzőekhez hasonlóan kerül rá a vezérlés. Belátható, hogy a gyorsítótárak mérete és a keresések időigénye igen lényeges a program futtatási sebességének szempontjából. Ezért például a Valgrind keretrendszerben itt assembly nyelven írt részletek is találhatók, a sebesség további növelésére. Az 1.2. ábrán látható a fordítási tábla és a gyorsítótár elhelyezkedése, illetve a fordításokat futtató processzor.



1.2. ábra. *Fordítási tábla és gyorsítótár*

Árnyékértékek

Amint már korábban röviden bevezettem, az árnyékértékek használatával a regiszterekben és a memóriarekeszekben lévő adatokról tárolhatunk valamilyen többletinformációt. Ezek működésének és használatának néhány részletét ismertetem a következőkben.

A Valgrind minden regiszterhez biztosít egy árnyékregisztert, amelyek a normál regiszterekkel teljesen egyenértékűen használhatóak a köztes reprezentáció utasításaiban. Mivel az IR szintjén az árnyékregiszterek semmiben nem különböznek a normál regiszterektől, azokon is ugyanazok az utasítások hajthatóak végre. Ennek köszönhetően az árnyékregisztereken végzett bonyolultabb műveletek megvalósítása sem okozhat problémát. A keretrendszer a regiszter allokáció során is egyenlően kezeli a normál- és árnyékregisztereket. Ezzel nagyobb teljesítmény érhető el, hiszen az árnyékregiszterek értékét is tárolhatják a processzor valódi regiszterei.

A regiszterekhez hasonlóan a memóriarekeszekhez is tartoznak árnyékértékek, amelyek az árnyékregiszterekhez hasonlóan használhatók. Ezeket természetesen a memóriához használható utasításokkal (load/store) érhetjük el a köztes reprezentációból.

A következőkben egy Valgrind-re épülő memóriakezeléssel kapcsolatos hibákat feltáró DBA eszközt mutatok be.

1.3.3. Valgrind – Memcheck

A Memcheck[8, 9] egy DBA eszköz, amely az analizált programokban a memóriával kapcsolatos hibákat képes felfedezni. A Memcheck által elvégzett ellenőrzéseket a következő pontokban mutatom be:

Címezhetőség vizsgálat Minden memória-hozzáférésnél ellenőrzésre kerül, hogy az adott program az adott memóriaterületet jogosan használhatja-e, azaz a kérdéses memóriaterület a program „tulajdonában” áll vagy sem. A nem címezhető terület olvasása, de főként írása súlyos következményekkel járhat. A Memcheck ezért folyamatosan követi például a dinamikus memóriaallokáció működését, és ennek megfelelően tudja meghatározni, hogy mely területek címezhetők. Hibás címzés észlelésekor a Memcheck jelzi a hibát.

Heap kezelés A dinamikus memóriakezelés sok hiba forrása lehet, ezért a Memcheck követi a programban allokált memóriaterületeket, és képes jelezni például a többszörös felszabadításokat, vagy a memóriaszivárgást (a felszabadítás elhagyását).

Másolási átlapolás Az átlapolódó memóriaterületekkel meghívott C könyvtári buffer másoló függvények (pl. `memcpy()`) végrehajtása nem definiált működést eredményezhet. Mivel ez nem kívánatos (és valószínűleg nem is szándékos), a Memcheck az ilyen esetekben is hibát jelez.

Definiáltság ellenőrzés A Memcheck a program által használt minden memóriarekesz és regiszter esetén követi annak definiáltságát, azaz, hogy a program során megtörtént-e azok inicializálása, azaz például konstans vagy definiált értékekből számított érték

beírása. Az ellenőrzés ebben az esetben a definiálatlan értékek felhasználását keresi, és jelzi hibaként. A Memcheck lényeges előnye a hasonló megoldásokhoz képest, hogy a definiáltságot bitenként követi és vizsgálja, azaz alkalmas például a bitmezők egyes bitjeinek követésére is.

A Memcheck a Valgrind által biztosított felműszerezési lehetőségeket legnagyobb részben a címezhetőség és a definiáltság követésénél használja ki. A korábban bemutatott árnyékértékeknek is ezekben az esetekben van kiemelkedő szerepe. A Memcheck minden memóriarekeszhez és regiszterhez egy azonos méretű árnyékértéket rendel, amelynek bitjei jelzik az eredeti érték bitjeinek definiáltságát. Ezen felül a memória minden egyes bájtját egy további bit jellemzi, amely a címezhetőséget adja meg. A felműszerezés szempontjából lényegében minden utasítást követni kell, hiszen a valódi adatokon végzett műveletek mellett az azokat leíró árnyékértékeket is frissíteni kell.

A dinamikus bináris felműszerezés nagyon előnyös ilyen jellegű feladatok megoldásához, hiszen a köztes reprezentációban felműszerezett kód a visszafordítás után natívan hajtódik végre a processzoron, azaz a felműszerező utasítások nem okoznak a szükségesnél lényegesen nagyobb sebesség-csökkenést.

A Memcheck működése

Ahogy a fejezet elején említettem, a Memcheck a Valgrind által biztosított árnyékértékeket a címezhetőség és a definiáltság információk tárolására használja fel. Ennek megfelelően tehát minden memória (és regiszter) bithez tartozik egy árnyékbit, amely a valódi bit definiáltságát mutatja. A program működése közben elvégzett műveletek nagy része általában valamilyen bemenő értékek (paraméterek) alapján állít elő egy új értéket. Hogy a definiáltság követhető legyen, minden ilyen művelet esetén a megfelelő árnyékértékeket is fel kell dolgozni és egy új értéket előállítani. Ezeket az extra műveleteket a felműszerezési lépésben kell hozzáadni a programhoz. Ügyelni kell arra, hogy a felműszerező utasítások kellően hatékonyak legyenek, hiszen szinte az összes eredeti utasítást követni kell. Látható, hogy ez a megközelítés jelentős többlet utasítást eredményez, valamint a program által felhasznált memória is (elvileg) kétszeresére növekszik.

A program futtatásának kezdetén a regiszterek – a stack pointer kivételével – definiálatlan állapotúak. Ezzel ellentétesen, az indításkor elérhető memóriaterületek (kód, globális változók) definiált állapotban vannak. A futás során a Memcheck követi a memóriakezeléssel kapcsolatos eseményeket is, mint például a dinamikus memória allokációk vagy a stack változásai. A malloc utasítással lefoglalt memóriaterület az allokáció után a Memcheck címezhetőként és definiálatlan értékkel tarja számon. A memóriaterületek felszabadítása után azokat a Memcheck ismét definiálatlan állapotba állítja. Ez a megoldás logikus, hiszen a kérdéses területet már egy másik alkalmazás felhasználhatta. A stack növelésekor (pl. függvényhívás) az újonnan használható címek szintén címezhetőek és definiálatlanok lesznek.

A program utasításainak végrehajtása során a különböző műveletek eltérő felműszerezést (azaz az árnyékértékek kezelését) igényelnek. Az egyszerű adatmozgató utasítások

esetén egyszerűen az adatokhoz tartozó árnyékértékeket kell a megfelelő árnyékregiszterek vagy memóriarekeszek között mozgatni, a valódi adatokkal párhuzamosan. A bonyolultabb műveletek esetén akár az adatok figyelembe vételére is szükség lehet. Egy ilyen utasítás például a bitenkénti-ÉS művelet, ahol egy definiált állapotú és nulla értékű operandus esetén a másik operandus definiáltsága nem számít, hiszen a kimenet biztosan nulla lesz. Más műveletek felműszerezéséhez eltérő eljárásokra van szükség, azonban a bemutatottakból is látható a működés alapvető körvonala.

A Memcheck működésével kapcsolatban még egy érdekes kérdés, hogy a nem definiált értékekkel végzett műveletek esetén azonnal szükséges-e a probléma jelzése, vagy elegendő néhány speciális esetben elvégezni az ellenőrzést. A túl gyakori ellenőrzés nagymértékben lelassíthatja a program futását, viszont a hibák túl késői megtalálása megnehezítheti a probléma eredeti forrásának azonosítását. A Memcheck ezen szempontok alapján csak néhány kiemelt esetben végez ellenőrzést a felhasznált értékek definiáltságára. Ezek között megtalálhatóak a memória-hozzáférésekben felhasznált címek, a feltételes utasítások feltételei és a rendszerhívások paraméterei. Megjegyzendő, hogy a nem definiált értékekkel végzett műveletek nem feltétlenül okoznak hibát.

Ahogyan tehát látható, a Memcheck egy igen jól használható eszköz a programokban előforduló memóriakezeléssel kapcsolatos hibák feltárására.

1.3.4. Valgrind – Callgrind

A Callgrind[10] a Valgrind keretrendszerben működő dinamikus bináris analízis eszköz, amely a vizsgált program függvényhívási kapcsolatait feltérképezve képes a hívási gráf létrehozására. Ahogyan az 1.1.1. fejezetben láthattuk, a hívási gráfokat legalább kétféle módon ábrázolhatjuk. A két lehetőség a kontextus érzékeny, illetve érzéketlen megoldás. A Callgrind mindkét fajta gráfot képes létrehozni, a felhasználó a megfelelő beállítási lehetőséggel választhat. Itt egészen pontosan az adható meg, hogy a Callgrind milyen mélységig kövesse kontextus érzékenyen a hívási kapcsolatokat. Az alapértelmezett beállítás esetén kontextus érzéketlen gráfot kapunk.

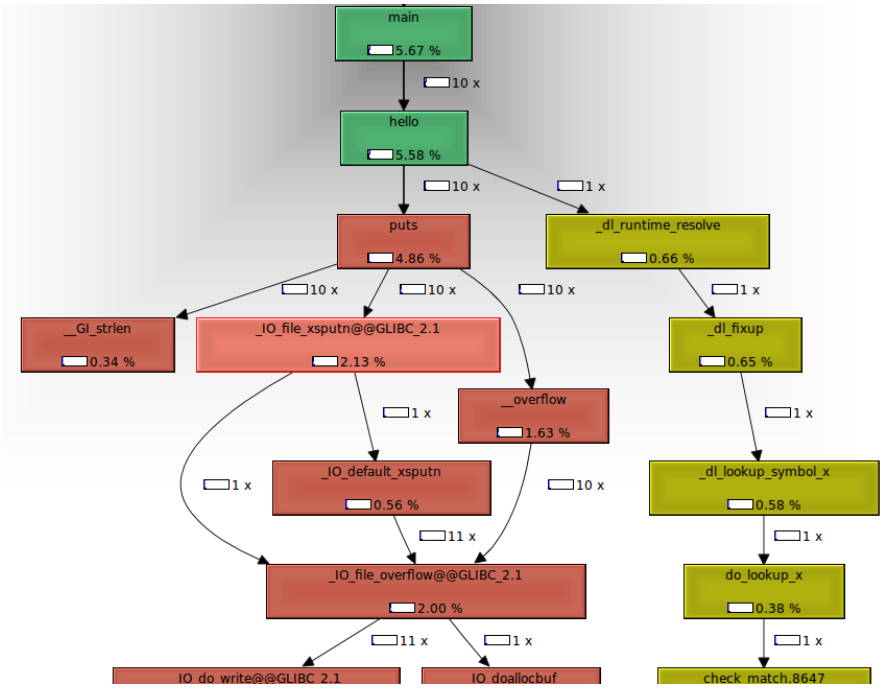
A Callgrind fontos képessége, hogy a hívási gráf alapján képes bizonyos „költségek” továbbterjesztésére a függvények között. Ez azt jelenti, hogy a hívási kapcsolatban lévő függvények esetén a hívott költségeit hozzáadja a hívó összköltségéhez, míg végül a program legkülső függvénye az összes költséget tartalmazni fogja. Ezzel a megadással minden függvényhez két költségadatot rendelhetünk, az egyik a függvény saját költsége, amely nem tartalmazza az általa hívott rutinok költségeit, a másik pedig az összesített költség, amely tartalmazza ezeket is. Ez a megadás hasonlít a gprof esetén bemutatotthoz (l. 1.3.1. fejezet), azonban míg ott a továbbterjesztett adatok a futási idők voltak, ezek a Cachegrind esetén a költségek.

A Callgrind által használt „költségek” – a vizsgálat céljától függően – sokféle információt hordozhatnak. Egyszerűbb esetben a költség lehet a végrehajtott utasítások száma vagy például az adat olvasások száma. A Callgrind-et kiegészítő Cachegrind ezen felül a cache szimulációjával képes a program és a cache memóriák együttműködését vizsgálni.

A költségek ebben az esetben lehetnek a különböző szintű gyorsítótár hozzáférések. A Cachegrind két elsőszintű (adat és utasítás), valamint egy második szintű közös gyorsítótárat feltételezve végzi a szimulációt. Ez megfelel az elterjedt számítógépekben használt felépítésnek[11].

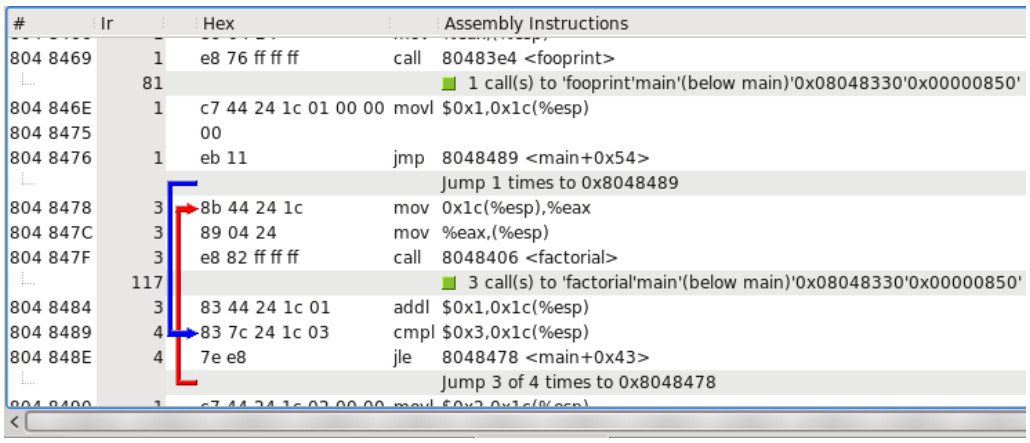
A Callgrind és a Cachegrind által gyűjtött adatok grafikus formában is megtekinthetők a KCachegrind programmal, amely a hívási kapcsolatok és a költségek megjelenítésén túl képes az egyes költségeket a program utasításlistájához vagy, amennyiben rendelkezésre áll, annak forráskódjához is kapcsolni.

Egy a Kcachegrind által kirajzolt hívási gráf látható az 1.3. ábrán. Megfigyelhetőek a csomópontokkal reprezentált függvények esetén a költség értékek, amelyek itt százalékosan vannak megadva, valamint az éleken (azaz hívási kapcsolatokon) lévő számláló értékeket. A költség ezen az egyszerű példán az utasításfelhozatalok száma. Ez tulajdonképpen az egyes függvényekben található assembly utasítások futásainak a száma. Látható, hogy az ábrázolt hívási fa kontextus független.



1.3. ábra. *KCachegrind hívási gráf példa*

Az 1.4. ábrán a költségeket a program assembly utasításaihoz rendelve láthatjuk. Az egyes sorokban szerepelnek a végrehajtott utasítás memóriacímén kívül a végrehajtás költsége és az utasítás gépi kód valamint assembly utasítások formájában. A költségek itt is az utasításfelhozatalok számát jelentik. Érdemes megfigyelni, hogy a függvényhívó `call` utasítások esetén a költség jóval nagyobb, mint a többi esetben. Ezt az okozza, hogy a KCachegrind itt az adott függvényhívás teljes költségét jeleníti meg, azaz azoknak az utasításoknak a számát, amelyek a meghívott függvényben lefutnak. A feltétel nélküli és feltételes ugró utasításoknál a végrehajtások száma mellett az is megjelentésre kerül, hogy a feltétel hányszor teljesült, azaz hány tényleges ugrás történt. Amint láthatjuk, a Callgrind és a megjelenítést végző KCachegrind egy igen jól használható eszközkészlet a programok



1.4. ábra. *KCachegrind assembly lista nézet*

vizsgálatára, amely többféle szempont szerint (Cachegrind) is képes analizálni a működést. Miután az eszköz a Valgrind keretrendszert használja, a futtatás kellően gyors, és a program módosítására sincs szükség.

A Callgrind működésének és használatának részletesebb bemutatását itt nem fejtem ki.

1.4. Dinamikus adatfolyam gráfok használata

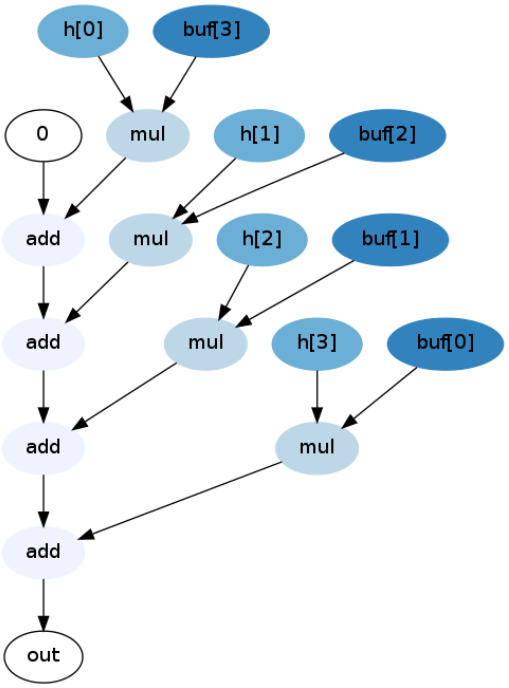
Ebben a fejezetben a programok analizálásának egy további aspektusát mutatom be, a dinamikus adatfolyam gráfokat. A dinamikus adatfolyam gráfok lényege, hogy használatukkal megjeleníthetőek a program adatáramlásai a futás közben. Ez azt jelenti, hogy például az egyes aritmetikai utasítások által kiszámított értékek további felhasználásai, illetve a műveletek paramétereinek forrásai leolvashatók.

A dinamikus adatfolyam gráfok irányított gráfok[12], amelyekben a csomópontok jelölik a valamilyen „új” adatot előállító utasításokat, míg az élek a tulajdonképpeni adatok terjedését mutatják a csomópontok között. Mivel minden ábrázolt utasításban előáll egy új adat, minden csomópontból indulhatnak ki élek. Ezek az élek azokba a csomópontokba mutatnak, amelyek az előállított adatot felhasználják. Egy adat felhasználása megfelel annak, hogy egy következő utasítás bemeneti paraméterként fogadja.

A dinamikus adatfolyam gráf a program vezérlési utasításaira (elágazásokra, ciklusokra) alapvetően nem tartalmaz információt, azonban az adatfolyamok megjelenítésével így is nagyon kifejező tud lenni. A program minden egyes utasításához annyi különálló csomópont tartozhat, ahányszor az adott utasítás új értéket állított elő a futás során. Ez jellemzően megegyezik a futások számával.

A dinamikus adatfolyam gráfok minden esetben irányított körmentes gráfok, ami abból következik, hogy egy új adat előállításához természetesen nem lehet magát az új adatot, vagy annak leszármazottját is felhasználni. Ezen gráfelméleti tulajdonság kihasználása remélhetőleg a későbbiekben megkönnyíti majd a feldolgozást, az ilyen tulajdonságú gráfokon végezhető topológiai rendezés vagy a kritikus utak módszerének használatával.

Az 1.5. ábrán egy négy fokszámú FIR szűrő egyetlen kimeneti mintájára vonatkozó adatfolyam gráf elképzelt részlete látható. Az „add” illetve „mul” feliratú csomópontok az



1.5. ábra. *FIR szűrő elképzelt adatfolyam gráfja (egy kimeneti mintához)*

összeadási és szorzási műveleteket jelentik. A „h" tömb elemei a szűrő együtthatóit tárolják, míg a „buf" a bemeneti késleltetővonalat jelképezi. Megfigyelhető a FIR szűrőkre minden esetben jellemző MAC (multiply and accumulate) műveletek sorozata. A gráfnak megfelelő működést megvalósító egyszerűsített kódrészlet az 1.1. listán látható. Az egyszerűbb ábrázolhatóság miatt az ábrán nem szerepelnek a szűrőegyütthatók és a bemeneti adatok beolvasásához szükséges műveletek, valamint a ciklusszámláló sincs megjelenítve. A teljes adatfolyam gráfon látható lenne az index változó növelése, valamint a memóriaműveletek esetén a báziscím plusz index jellegű címzés.

1.1. lista. *FIR szűrőt megvalósító kódrészlet*

```
int i;
float out = 0;
...
for(i=0, i<4; i++){
        out += h[i] * buf[3-i];
}
...
```

Amint az ábrán látható a vezérlési szerkezet lényegében semmilyen formában nem jelenik meg a gráfon, mindössze az ismétlődő csomópontok adhatnak erre némi információt. A forráskódot és az ábrát összehasonlítva látható, hogy a ciklus belsejében található szorzás és összeadás a végrehajtásnak megfelelően négyszer jelenik meg, annak ellenére, hogy a programban csupán egyszer szerepelnek.

A dinamikus adatfolyam gráfokról tehát leolvasható, hogy a program végrehajtása során mely utasítások mely másik utasítások által előállított adatokat használják fel, azaz az adatok forrásait és áramlási útjait.

1.4.1. Valgrind – Redux

A Redux[12] egy Valgrind alapú DBA eszköz, amely képes a vizsgált programok dinamikus adatfolyam gráfjainak felvételére. A Valgrind által biztosított felműszerezési lehetőségeket kihasználva a Redux – a megszokott módon – a vizsgált program bármilyen módosítása nélkül működik.

A keretrendszer által megvalósított árnyékértékek a Redux működésében nagy szerepet kapnak. A működés lényege, hogy minden regiszterben vagy memóriában lévő adathoz tartozó árnyékérték egy pointert tartalmaz, amely az ahhoz tartozó adatfolyam gráf csomópontra mutat. A gráf felépítésénél a kapcsolatok mindig egy adat felhasználása felől mutatnak az adat forrása felé, azaz minden node referenciákat tárol azokra a node-okra, amelyek adatát felhasználja. Az adatok „áramlása" természetesen fordított irányú, azonban a gráf felépítésének szempontjából ez a praktikusabb megoldás.

Az adatfolyam gráf felépítése a program futásával párhuzamosan történik. A vizsgálat kezdetén a gráf üres, az összes árnyékregiszter és árnyékmemória egy speciális node-ra mutat, amely jelzi, hogy a vizsgálat során még nem került az adott regiszterbe vagy memóriacímre semmilyen adat. A futás során a felműszerezés minden új adatot előállító (pl. aritmetikai) utasítás esetén egy új csomópontot ad a gráfhoz. Az új adat kiszámításához felhasznált operandusokhoz tartozó node-ok referenciái bekerülnek a létrehozott csomópontba, azaz eltárolódnak a források.

A gráf csomópontjai tulajdonságokkal rendelkeznek. A topológia szempontjából legfontosabb az adott adat kiszámításához felhasznált forrás-adatokra mutató referenciák, azonban itt található még az elvégzett művelet típusa és a művelet eredménye, azaz a regiszter vagy memória aktuális értéke. A referenciák tulajdonképpen a gráf éleit reprezentálják.

A felműszerezés során beszúrt utasítások feladata az adatfolyam gráf folyamatos építése a program működésével párhuzamosan. Aritmetikai műveletek esetén például a felműszerezés egy új csomópontot ad a gráfhoz, amelyben rögzíti az adott értéket, az annak kiszámításához felhasznált csomópontokra mutató referenciákat, valamint a művelet típusát. Egyszerű adatmozgató utasítások esetén a Redux nem hoz létre új csomópontot, mindössze a megfelelő árnyékértékeket (azaz a gráf csomópontjaira mutató pointereket) másolja a valódi értékekkel párhuzamosan. Ez a gráf méretét csökkenti, és az elhagyott adatmozgatások bizonyos szempontból nem is hordoznak információt.

A Valgrind (és így a Redux is) alapvetően 32 bites szavakhoz rendeli az árnyékértékeket. Ez az esetek nagy részében nem okoz problémát, azonban a 16 vagy 8 bites műveletek követése nehézséget okozhat. A Redux az ilyen esetekben speciális csomópontokat illeszt be, amelyek a szavas adatokból vágnak ki bájtokat, illetve bájtos adatokat fűznek össze 32 bites szavakká. Ennek a megoldásnak az előnye, hogy a keletkező gráf méretét nem növeli nagy mértékben, mégis követhetővé teszi a bájtos és félszavas műveleteket. Hátrány, hogy a kivágások és összefűzések megfelelő elhelyezése a felműszerezés során igen komplikált. További lehetőség lenne a memóriarekeszek és regiszterek tartalmát bájt szinten követni, így elhagyhatóak a komplikált kivágó és összefűző műveletek, azonban ez a gráf méretét olyan mértékben növelné, ami ezeket az előnyöket nem ellensúlyozná.

A programban előforduló konstansokhoz szintén egy gráf csomópont rendelhető. Ezek a csomópontok abban a tekintetben speciálisak, hogy a bennük szereplő adatnak nincsenek forrásai, hiszen itt nem valamilyen más adatok alapján kiszámított, új értékekről van szó. Ha a programban a fent említett nem inicializált értékre való hivatkozást látunk, feltételezhetjük, hogy a programban valamilyen hiba van, hiszen elsőként minden esetben kezdőértéket kell adni a felhasznált változóknak.

A gráf felépítésének módjából adódóan minden időpontban csak azok a csomópontok érhetők el (járhatók be), amelyek az aktuálisan memóriában illetve a regiszterekben tárolt adatok kiszámításához felhasználásra kerültek. Elképzelhetőek olyan esetek, amikor a program egy bonyolult számítás eredményét csupán vezérlési célokra (pl. feltételként) használja. Ekkor szükséges lehet extra élek hozzáadására, amelyek az így létrejövő kapcsolatokat jelzik, illetve meghagyják a számításhoz tartozó csomópontok bejárásának lehetőségét.

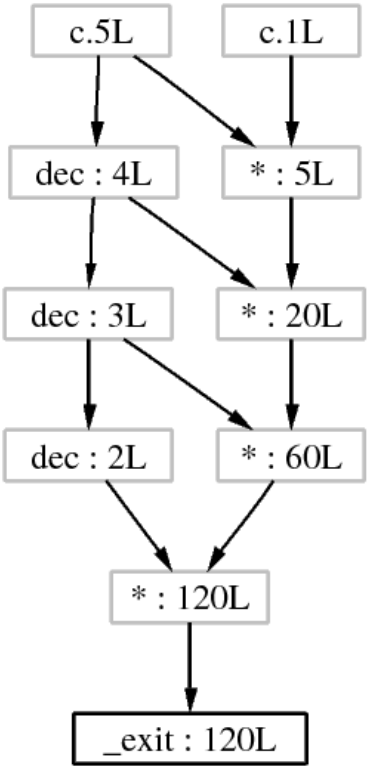
A csomópontok tárolása a memóriában lefoglalt blokkokban történik. A már nem elérhető csomópontok lényegében feleslegesen foglalják a memóriát. Ezeket a területeket valamilyen szemétgyűjtő (garbage collection) eljárással fel lehetne szabadítani, azonban a Redux ezt nem alkalmazza, mivel a vizsgálható méretű programok esetén nem jelent szűkös erőforrást a memória.

A Redux a program kimeneteit a rendszerhívásokon keresztül rögzíti. Ezekben az esetekben a felhasznált regiszterekhez és memóriarekeszekhez tartozó gráf csomópontok elmentésre kerülnek. Később ezeken a csomópontokon elindulva és bejárva a gráf elérhető részét, a program kimeneteinek létrehozására jellemző képet kaphatunk. A rendszerhívások gyakran memóriaterületekre mutató pointereket várnak bemenetként. Az ilyen módon átadott paramétereket a Redux az eddig bemutatott módon nem tudja követni. Itt a Valgrind biztosít további segítséget azáltal, hogy a rendszerhívások memóriaműveleteit továbbítja a Redux felé. Ezek alapján már az összes átadott adat követhető és ábrázolható.

Az analizált program lefutása közben jelentős mennyiségű csomópont jön létre az adatfolyam gráfban. Ezek mindegyikének megjelenítése teljesen átláthatatlan ábrákat eredményezne, illetve a node-ok jelentős része nem hordozna információt a program működésére nézve. Ennek a problémának a megoldására a Redux csak a program külvilággal való kapcsolódásait, azaz a rendszerhívásokat veszi figyelembe. Csak azokból a csomópontokból elérhető gráf részek jelennek meg a végeredményen, amelyek a rendszerhívásokban paraméterekként szerepelnek. Ez a megoldás logikus, hiszen a program külvilág felé átadott adatait tekinthetjük a program működésének eredményeként. A Redux az adatfolyam gráfokon további egyszerűsítéseket is végez. Így például a kevés alkalommal felhasznált adatokat elhagyhatja, illetve bizonyos utasítások láncait összevonhatja.

A vizsgálat eredményeként kapott adatfolyam gráfot a Redux nem közvetlenül képként, hanem a csomópontokat és a köztük lévő éleket leíró *dot* fájlként adja meg. Ez a fájlt ezután a Graphviz program használatával alakítható ténylegesen megjeleníthető képfájl formátumba. A Redux kimenetére egy példa az 1.6. ábrán látható.

A Redux készítői a programot (és az adatfolyam gráfokat) többféle feladat és probléma megoldására javasolják. Az egyik ilyen használati eset a programok által végzett számítások „lényegének” kinyerése, és ez alapján programok összehasonlítása, hasonlóságok ke-



1.6. ábra. *Redux által generált adatfolyam gráf iteratív faktoriális számításhoz[12]*

resése. Egy számítás „lényege" azt fejezi ki, hogy a program milyen úton jut el egy adott eredményhez, a vezérlési szerkezetektől és egyéb tényezőktől eltekintve. Ez alapján eltérő megvalósítású (esetleg eltérő nyelven írt) programok hasonlíthatóak össze, és bizonyos esetekben meglepő hasonlóságok fedezhetők fel. Miután a Redux csak a külvilág felől érzékelhető kimenetek előállítását végző részgráfokat ábrázolja, a hasonlóságok még jobban láthatóak lehetnek.

Másik igen ígéretes felhasználási lehetőség a programok fejlesztése közbeni hibakeresés területén adódik. A hagyományos hibakeresés során ha egy számítás eredménye hibás, nehezen követhető vissza annak előtörténete. A dinamikus adatfolyam gráfok pontosan erre adnak egy jól használható eszközt. Így egy hibás érték esetén mindössze az abból elérhető csomópontokat kell ábrázolni, és így láthatóak lesznek a hibás érték kiszámításához vezető utasítások. A Redux esetén itt még az egyes rész-számítások eredményei is megfigyelhetőek, ami tovább egyszerűsítheti a hiba okának megtalálását.

Egy érdekes lehetősége a dinamikus adatfolyam gráfoknak a vizsgált program párhuzamosíthatóságára vonatkozó következtetések levonása. Ennek a lényege, hogy miután az adatfolyam gráf megmutatja a program egyes műveleteinek egymáshoz való kapcsolódását, ez alapján megtalálhatóak az erősen összefüggő és az egymástól független részek. Ezzel a témakörrel részletesebben a dolgozatom további fejezeteiben foglalkozom.

A dinamikus adatfolyam gráfok további alkalmazásai lehetnek még a programok működésének visszafejtését illetve jobb megértését célzó feladatok.

A Redux forráskódja elérhető, azonban futás közbeni kipróbálására tett próbálkozásaim nem voltak sikeresek. Problémát okoz, hogy a 2005-ben írt forráskód függőségei között a glibc akkor aktuális változata szerepel, ami a kód jelenlegi lefordítását nagyban megnehezíti. További probléma, hogy a kód erősen prototípus szintű, azaz a működése ennek megfelelően nem túl megbízható[13]. Mindezek ellenére, a Redux alapelveinek megértése után, azokból kiindulva a dinamikus adatfolyam gráfok létrehozásával kapcsolatos feladatok valamivel könnyebben hajthatók végre.

2. fejezet

Algoritmusok párhuzamosíthatósága

Ahogy a bevezetőben már felvázoltam, a manapság egyre jobban elterjedő többprocesszoros rendszereket a nem párhuzamosított algoritmusok nem tudják megfelelően kihasználni. Természetesen vannak algoritmusok, amelyek eredendően jól párhuzamosíthatók, azonban vannak olyanok is amelyek nem rendelkeznek ezzel a tulajdonsággal. Ilyen algoritmusok például a stream jellegű videó vagy hang feldolgozást végző eljárások. Ha lehetne találni egy eljárást, amellyel ezeket az algoritmusokat is párhuzamosíthatnánk, az – főleg a beágyazott rendszerek esetén – a hatékonyság nagymértékű növekedését eredményezhetné. További lehetőség a jelenleg csak szoftveres megvalósítások szétbontása szoftverben és hardverben megvalósított részekre. Ennek implementációjához például FPGA-k és azokban létrehozott soft- vagy hard-core processzorok használhatók.

2.1. Utasítás szintű párhuzamosíthatóság

A programok, amelyekre legtöbbször utasítások egymásutáni sorozataként gondolunk, az utasításaik szintjén is párhuzamosíthatók. Ez azt jelenti, hogy a program végrehajtása során az utasítások közül egy időben nem csak egy, hanem több is futtatható a program működésének megváltoztatása nélkül. Ez egy több processzort tartalmazó rendszerben azonos a nagyobb feldolgozási teljesítménnyel. Természetesen az utasítások párhuzamosításának vannak korlátai, amelyek akadályozhatják ezeket a lehetőségeket.

Az utasítás szintű párhuzamosíthatóságot alapvetően az utasítások között fennálló függőségek korlátozhatják[14]. Ezek a függőségek több forrásból is eredhetnek, éppen ezért feloldásuk nem minden esetben lehetséges. A függőségeket kategorizálhatjuk típusuk szerint. Így megkülönböztethetünk valódi, hamis, kimeneti és vezérlési összefüggéseket. Ezeknek a függőségeknek a jellemzőit a következőkben ismertetem.

Valódi függőségek A valódi függőségek – amint az elnevezés is mutatja – a program működéséhez elengedhetetlen kapcsolatok. Ezekben az esetekben egy utasítás kimenetét egy következő utasításban használjuk fel. Mivel ezek a kapcsolatok adják a program működését, a köztes eredményt szolgáltató utasításnak mindenképpen az eredményt felhasználó utasítás előtt kell lefutnia, ellenkező esetben az algoritmus működése nem lesz helyes.

Hamis függőségek A hamis, illetve kimeneti függőségek, amelyeket tárolási függőségeknek is nevezhetünk[15], nem valós kapcsolatok, azok mindössze a korlátozott tárolási lehetőségek (regiszterek száma, memóriakapacitás) miatt alakulnak ki. A probléma lényege, hogy a program működése során egy tárolórekesz, például egy regiszter, egymás után többféle értéket is tárolhat. Az utasítások párhuzamosítása esetén valahogyan biztosítani kell, hogy minden utasítás a neki megfelelő állapotban találja a tárolórekeszt. Ehhez lényegében az utasítások közötti szinkronizáció szükséges, amely megakadályozza például, hogy egy utasítás felülírjon egy másik által előállított, de még nem felhasznált értéket. A tárolási függőségek triviálisan feloldhatóak lehetnek a regiszterek „átnevezésével", azaz ha egy regiszterben csak egyetlen adatot tárolunk. Mivel ehhez nagyon nagy számú regiszterre lenne szükség, csak a közelítő megoldás jöhet számításba. Ekkor a rendelkezésre álló regiszterek közül kell minden utasításhoz és előálló értékhez azt kiválasztani, amelyre nem állnak fenn ilyen jellegű függőségek.

Vezérlési függőségek A vezérlési függőségek, hasonlóan a valódi függőségekhez, a program működésének alapjaiból erednek. Ezekben az esetekben bizonyos kódrészletek lefutása függ valamilyen számítások eredményétől, így a párhuzamosítás vagy a sorrend felcserélése itt sem lehetséges. A vezérlési függőségek kiküszöbölése a legtöbb esetben nem lehetséges, hiszen egy feltételes ugráshoz érve a végrehajtásban, a folytatáshoz ismernünk kell a feltételben szereplő kifejezések értékét.

A vezérlési függőségek a valódi függőségeknél jóval erősebb korlátot jelentenek a párhuzamos futtatásra. Ennek az az oka, hogy amíg a valódi függőségek csak néhány egymás utáni utasítás között állnak fenn, és csak ezek futtatását nem teszik párhuzamosíthatóvá, addig a vezérlési függőségek teljes kódblokkok[1] (basic block) esetén írnak elő feltételeket. Ezáltal a párhuzamosan történő futtatás legnagyobb részben csak az egyes blokkokon belül valósítható meg, ami által a kinyerhető párhuzamosság nagy része elvész[14]. A megoldást például a feltételek kiértékelését előrejelző branch predictor jelentheti, amellyel a végrehajtás az előrejelzett ágon folytatható. Ha később az előrejelzés helyesnek bizonyul, a végrehajtás folytatódhat tovább, és az eredmények tovább használhatók, azonban hibás előrejelzés esetén az eredményt el kell dobni és a másik ágat kell végrehajtani. Látható, hogy ebben az esetben a párhuzamosítás hatékonysága nagyban függ az előrejelzések pontosságától.

A programokban szereplő valódi és vezérlési függőségek megadnak egy felső korlátot az adott algoritmus párhuzamosíthatóságára, amelyet semmilyen módon nem lehet túllépni. Ez a korlát az algoritmus felépítéséből és működéséből adódik. Az algoritmusokban lévő valódi párhuzamosság általában jóval nagyobb mint a tényleges megvalósítás során kinyerhető párhuzamosság, hiszen a végrehajtás során az erőforrások (processzorok, regiszterek), még több processzoros rendszerekben is erősen korlátozottak.

[1] A kódblokkok (basic block) olyan egymás utáni utasításokat jelentenek, amelyeken a végrehajtás folyamatosan, utasításonként halad végig. Minden blokk egy belépési és egy kilépési ponttal rendelkezik. A határokat a feltételes vagy feltétel nélküli ugró utasítások jelentik.

2.2. Függvény és modul szintű párhuzamosíthatóság

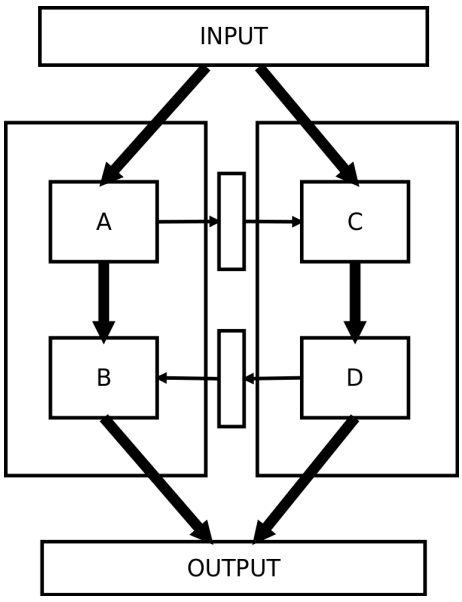
Amíg az utasítás szintű párhuzamosítás az egyes utasítások közötti függőségi kapcsolatokat figyelembe véve biztosíthatja a párhuzamosságot, elképzelhető az algoritmusok párhuzamos végrehajtása nagyobb egységenként is. Ekkor függvény vagy modul szintű párhuzamosításról beszélhetünk.

Az ilyen szintű párhuzamosíthatóság esetén is fellépnek a korábbiakhoz hasonló függőségek, amelyek korlátozzák a lehetőségeket. Ilyen függőségeket jelentenek a függvényhívások, amelyek esetén a hívó függvény addig nem használhatja fel a hívott által visszaadott eredményt, amíg a hívott függvény vissza nem tér, így a párhuzamos végrehajtás nem lehetséges. A függvények a hívási kapcsolatokon kívül globális változókon keresztül is adhatnak át egymásnak adatokat, amelyek szintén a párhuzamosítást akadályozhatják.

Ha az algoritmusok részegységeit tekintjük, találhatunk olyan modulokat, amelyek egymástól függetlenül vagy csak minimális egymásra hatással futnak le. Belátható, hogy amennyiben ezeket a lazán kapcsolódó részegységeket külön processzorokon hajtjuk végre, akkor az a teljes működést gyorsabbá teheti.

Ehhez a fajta szeparációhoz az algoritmus vizsgálata után fel kell tárni az egyes részegységek (és függvények) kapcsolatait. A részegységek között átáramló adatmennyiség meghatározása és az adatfolyam időbeli eloszlása is érdekes információkkal szolgálhat. Amennyiben például két részegység között a program indításakor nagy mennyiségű adatáramlás látható, a futás során azonban nincsen kommunikáció, a két egység üzemeltethető párhuzamosan is.

Természetesen a párhuzamosítás után létrejövő egymástól függetlenül futó szálakat valamilyen szinkronizációs eszközök használatával a kritikus pontokon egymáshoz kell igazítani.



2.1. ábra. *Modul szintű párhuzamosítás megvalósítása*

A 2.1. ábrán az ilyen módon történő párhuzamosítás megvalósítását szemléltetem. Az ábrán a nyilak vastagsága a részegységek közötti kapcsolatok szorosságát jelzi. Amint lát-

ható, az A és C valamint a B és D egységek között laza a kapcsolat. Ennek megfelelően a végrehajtás szétválasztható két végrehajtó egységre, amelyek közül az egyik az A és B, míg a másik a C és D alegységeket hajtja végre. A két processzoron futó egységek közötti kommunikáció megfelelő szinkronizálását biztosítani kell ebben az esetben is.

2.3. Analízis lehetőségek

A programokban lévő párhuzamosítási lehetőségek megtalálásához tehát elsőként az egyes utasítások, függvények és részegységek közötti függőségeket kell megtalálnunk. Ehhez használhatóak statikus és dinamikus analízis eszközök, mint például a hívási fák vagy a dinamikus adatfolyam gráfok. A különféle (statikus, dinamikus) hívási gráfok megmutathatják, hogy a program függvényei között milyen kapcsolatokat kell figyelembe venni a párhuzamosíthatóság vizsgálatánál.

A már bemutatott dinamikus adatfolyam gráfok használatával elsősorban az utasítások közötti valódi függőségeket lehet megjeleníteni és vizsgálni, azonban segítségével a nagyobb egységek közötti adatáramlások is megjeleníthetők. Ehhez például a függvényekbe bemenő és azokból kilépő adatokat kell csak figyelembe venni, a belső adatáramlások elhanyagolásával.

További lehetőség az algoritmusok vizsgálatára a vezérlési gráf felvétele. A vezérlési gráfok megmutatják, hogy a program kódblokkjai között milyen vezérlési függőségek állnak fent, azaz ezek alapján is következtethetünk a párhuzamosíthatósági tulajdonságokra.

A dinamikus függőségi gráfok[15] egyesítik a vezérlési gráfok és az adatfolyam gráfok által megjelenített függőségi kapcsolatokat, ennek megfelelően ezeken az ábrákon láthatóak a valódi és a vezérlési függőségek is.

A következőkben a feladat megoldását ismertetem, amely során bemutatom az aggregált adatfolyam gráfok vizsgálatát lehetővé tévő eszköz elkészítését.

3. fejezet

A feladat megoldása

Az eddigi fejezetekben leírtak alapján látható, hogy a profiling és azon belül a memória profiling igen szerteágazó terület, amely a párhuzamosítási lehetőségek vizsgálatához is megfelelő eszközöket használ. Ilyen eszköz lehet a hívási gráfok vagy a dinamikus függőségi gráfok felvétele, amelyek segítségével feltárhatjuk a program belső összefüggéseit, azaz az utasítások és függvények kapcsolatait, a fennálló függőségeket.

A hívási gráf megmutatja a program függvényeinek kapcsolatait. A párhuzamosíthatóság szempontjából a függvényhívások függéseket jelentenek, amelyek a hívó és a hívott szubrutint összekapcsolják. A hívási gráf elemzésével feltárhatóak a vizsgált program olyan alegységei, amelyek egyáltalán nem vagy csak keveset kommunikálnak egymással. Ha más függőség sem áll fenn, ezek a részek párhuzamosíthatók.

A dinamikus adatfolyam gráfok a program utasításai között fennálló adat függőségeket teszik láthatóvá. Az adat függőségek megjelenítése a függvényhívások vizsgálatánál erősebb eszköz, mivel az összes átadott adat (például globális változók vagy pointerek használata) rögzíthető. Az adatfolyam gráf hátránya, hogy az utasítás szinten történő vizsgálat esetén a gráf hatalmas mérete miatt annak megjelenítése és értelmezése szinte lehetetlen. Ennek a problémának a megoldását jelenti a rögzített információk aggregálása.

A feladat megoldása során létrehozott analízis eszköz, a dinamikus adatfolyam gráfok függvények és nagyobb egységek szerinti aggregálásával még hatékonyabb vizsgálatokat tesz lehetővé. Az aggregálás lényege, hogy a program egyes részegységei között áramló adatokat összesített formában megjelenítve, a gráf mérete kezelhető marad, és a függőségekre vonatkozó információk könnyedén kinyerhetők.

Az átadott adat mennyisége mellett az adatáramlás időbeli lefutása szintén lényeges a párhuzamosíthatóság vizsgálatához. Teljesen eltérő megközelítést igényel például egy a program futásának elején feltöltött és utána csak olvasott táblázat, illetve egy folyamatos kommunikáció két részegység között.

A megvizsgált szakirodalom és az elméleti megfontolások alapján az algoritmusok vizsgálatához felépített rendszert a következő fejezetekben mutatom be részletesen. A megvalósításhoz felhasznált eszközök és eljárások mellett az általam elkészített analízis eszközt is részletesen ismertetem.

A programok vizsgálata több, egymást követő lépésben valósítható meg. Mivel a dinami-

kus adatfolyam információk a program futása közben nyerhetők ki, a programot egy olyan környezetben kell futtatni, amely lehetővé teszi a végrehajtás követését utasítás szinten. A megoldás során ennek megfelelően a programot egy processzor szimulátor használatával futtattam, amely lehetővé tette a végrehajtás mélyreható követését. A 4.1. fejezetben ennek részletei olvashatók.

A futás folyamán kinyert információk feldolgozásához egy saját analízis eszközt készítettem. Ez az eszköz a kinyert információk feldolgozásával létrehozza a vizsgált program hívási gráfját és aggregált adatfolyam gráfját is. Az eszköz rövid specifikációját, magas szintű felépítését és megvalósításának részleteit az 5. fejezet tárgyalja.

Az analízis eszköz által létrehozott kimenetek könnyen értelmezhető, grafikus megjelenítéséhez különféle nyílt forráskódú eszközöket használtam, amelyek bemutatása a 4. fejezetben olvasható.

Az elkészített analízis eszköz kipróbálásához, és a párhuzamosíthatóság valós környezetben történő vizsgálatához egy JPEG kitömörítő algoritmust használtam. A vizsgált program bemutatása, az azon végrehajtott módosítások illetve a vizsgálati eredmények és azok értékelései a 6. fejezetben olvashatók.

4. fejezet

A vizsgálat környezete

Ebben a fejezetben az algoritmusok vizsgálatának környezetét, és a megvalósított analízis eszköz által használt formátumok sajátosságait mutatom be. Az implementáció részletes leírása – amely a következő fejezetben található – nagymértékben az itt leírtakra épül.

4.1. Processzor szimulátor

Ahogyan az előzőekben láthattuk, az algoritmusok működésének és felépítésének vizsgálatához valamilyen módon azokat futás közben kell vizsgálnunk. Ehhez a korábban bemutatott felműszerezési eljárások használhatóak, a fordító által segített felműszerezéstől a dinamikus bináris újrafordításig. A sokféle lehetőség közül a választás a Giano[16] processzor szimulátorra esett, amely választást a következők indokolták:

Hardware-software együttes szimuláció A Giano szimulátor képes a processzoros rendszer szimulációja mellett egy HDL nyelven megadott hardware rendszert is szimulálni úgy, hogy a két rendszer egymással szoros kapcsolatban áll. Ez a tulajdonság a távolabbi jövőben, az algoritmusok szeparálásánál lehet előnyös, ha az algoritmus egy részét software, másik részét hardware megvalósításban szeretnénk létrehozni.

Többprocesszoros rendszerek szimulációja A Giano alkalmas többprocesszoros rendszerek szimulációjára, ami a feladat eredeti elképzelésének megfelelő módon teszi lehetővé az algoritmusok vizsgálatából levont következtetések későbbi kipróbálását.

Sokféle rendszer-architektúra szimulációja A fentieken túl a Giano előnye még, hogy sokféle rendszer-összeállítás szimulálható vele. Többféle processzor, perifériák és buszrendszerek állnak rendelkezésre, illetve a rendszerek összeállítása grafikusan, könnyedén elvégezhető.

Nagyon részletes trace lehetőség Az algoritmusok analíziséhez fontos, hogy azok futása közben a lehető legtöbb információt tudjuk összegyűjteni azok működésről. A Giano nagy előnye ebből a szempontból, hogy apró módosítások árán az analízishez szükséges részletességgel biztosít információkat a program működésének minden aspektusáról.

A Giano által biztosított sokféle processzor közül egy ARM 7-es processzort választottam az algoritmusok vizsgálatához. Ennek a választásnak több oka is volt. Figyelembe vettem az utasításkészlet egyszerűségét, az ARM processzorok népszerűségét és gyakori alkalmazásukat beágyazott rendszerekben, illetve egyéb tényezőket.

Miután a szimulált processzor architektúrája nem egyezik meg az elterjedt PC-k architektúrájával, a vizsgált programok fordításához kereszt-fordítót kell alkalmazni, amely a PC-n futva képes az ARM processzoron működőképes gépi kódot előállítani. Ehhez a beágyazott rendszereknél gyakran használt Mentor Sourcery CodeBench[17] fordító ARM processzorokhoz szánt változatát használtam.

Az algoritmusok vizsgálatát a szimulált processzoron operációs rendszer nélkül végeztem. Az operációs rendszer használata ebben az esetben nehézkessé és jelentősen lassabbá tenné a szimulációt, valamint a keletkező trace is nagyrészt érdektelen utasításokat tartalmazna. A vizsgálati lehetőségek megvalósításához és az egyszerűbb algoritmusok vizsgálatához az operációs rendszer használata semmilyen egyéb okból sem indokolt.

4.1.1. A Giano rövid bemutatása

Ebben az alfejezetben a Giano szimulátort mutatom be érintőleges részletességgel[16]. A Giano egy Microsoft által kifejlesztett, forráskód szinten szabadon elérhető szimulációs keretrendszer, amely szinte tetszőleges rendszer szimulációjára használható. A Giano fejlesztése közben kiemelt szerepet kapott a hardver-szoftver együttes tervezés támogatása, így ilyen rendszerek szimulálása is lehetséges. Ennek az egyre elterjedtebb processzort és programozható logikát is tartalmazó rendszerekre való fejlesztés során van jelentősége.

A Giano a HDL nyelven írt hardver egységek szimulációjához képes külső HDL szimulátorokhoz kapcsolódni, és a teljes rendszer szimulációját elvégezni.

A Giano a szimulációk végrehajtásához lényegében csupán a keretrendszert biztosítja, a tényleges szimulációt az egyes modulok végzik. Ezek a modulok lehetnek a processzort, a buszrendszereket, a memóriát vagy egyéb perifériákat leíró egységek. Ebből a megközelítésből adódóan a Giano nagyon jól bővíthető és konfigurálható különféle rendszerek szimulációjához.

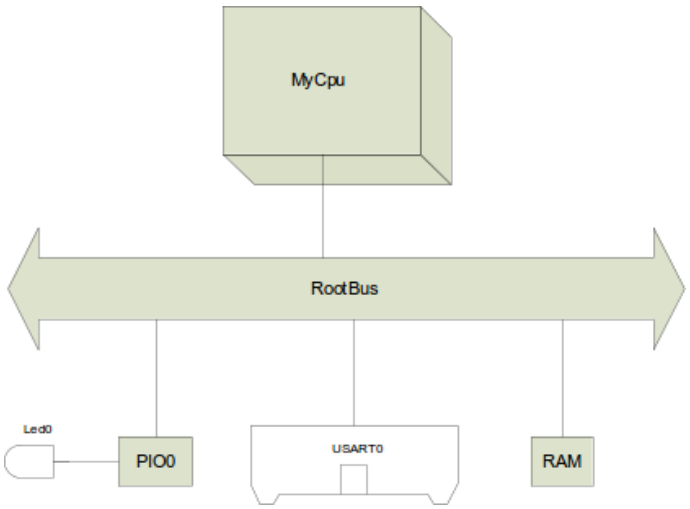
A szimulációkban használni kívánt modulokat és azok kapcsolatait egy PlatformXML nyelvű fájlban kell megadni. Ez a fájl XML formátumban tartalmazza a felhasznált modulokat, azok paramétereit és a köztük lévő kapcsolatokat. A Giano ezt a fájlt beolvasva építi fel a szimulációs rendszert. A konfigurációk megadása grafikusan is elvégezhető a Microsoft Visio használatával.

A Giano beépítetten tartalmaz többféle profiling eszközt, amely a program futása során képes az ilyen szempontból érdekes adatok gyűjtésére. A dolgozat következő részeiben a Giano által ilyen módon naplózott futások eredményeit fogom felhasználni az algoritmusok vizsgálatához.

4.1.2. A Giano használata és napló kimenete

A Giano számára – ahogyan már említettem – a szimulálni kívánt rendszer komponenseit és azok kapcsolatait egy PlatformXML nyelvű fájl írja le. Ebben a fájlban találhatóak továbbá a felhasznált részegységek paraméterei is. A feladat megoldásához az ARM 7-es processzor mellett mindössze egy RAM memóriaegységet szükséges a szimulált rendszerbuszra illeszteni. A rendszer – elsősorban a Giano kipróbálásához – tartalmaz még egy GPIO és egy UART egységet is. Ezek az egységek azonban a programok analíziséhez nem szükségesek.

A rendszer Microsoft Visio-ban megjelenő felépítése a 4.1. ábrán látható. Megfigyelhető a rendszerbuszra csatlakozó processzor (MyCpu), valamint a RAM memória.



4.1. ábra. *Giano rendszer felépítése*

A szimulálni kívánt rendszert leíró PlatformXML fájlban az egyes blokkok tulajdonságai találhatók meg. A 4.1. részleten a processzor jellemzőinek megadása látható. A processzor

4.1. lista. *A CPU jellemzői a PlatformXML fájlban*

```
<CPU name="MyCpu">
  <Property name="CpuType" value="ARM" />
  <Property name="ByteOrder" value="LittleEndian" />
  <Property name="Implementation" value="arm" />
  <Property name="BootAddress" value="0" />
  <ConnectsTo name="RootBus" />
</CPU>
```

típusa mellett megadható a bájt-sorrend valamint például az indulási cím, ahonnan a processzor indulásakor az utasítások végrehajtását elkezdi.

A memóriára vonatkozó beállítási lehetőségek a 4.2. listán figyelhetők meg. A RAM tulajdonságai közül a memória tartomány kezdőcímét (StartAddress) és méretét (Size), valamint a kezdeti tartalmat meghatározó bináris állományt (PermanentStorage) megadó sorokra hívnám fel a figyelmet. A RAM kezdeti tartalmának megadásával igen egyszerűen betölthető a futtatni kívánt kód, és az indítás után a processzor azonnal elkezdheti az utasítások végrehajtását.

4.2. lista. *A RAM jellemzői a PlatformXML fájlban*

```
<Memory name="RAM">
  <Property name="MemoryType" value="RAM" />
  <Property name="StartAddress" value="0" />
  <Property name="Size" value="2097152" />
  <Property name="Implementation" value="memory" />
  <Property name="PermanentStorage" value="hello.bin" />
  <ConnectsTo name="RootBus" />
</Memory>
```

A bemutatott részletek alapján látható, hogy a szimulálni kívánt rendszer összeállítása és a paraméterek megadása milyen egyszerűen elvégezhető.

A szimulált rendszeren futó program viselkedésének részletes naplózása a Giano egy fontos lehetősége. Naplózásra kerül a processzor által végrehajtott összes utasítás és az utasítások hatásai is. Az utasításokról a naplófájlokban a következő információk kerülnek rögzítésre:

Programszámláló
: Az aktuális programszámláló érték, azaz az adott utasítás memóriacíme.

Opcode
: Az aktuálisan végrehajtott utasítás műveleti kódja. A memóriában a programszámláló által meghatározott címen található érték.

Regiszterértékek
: A processzor regisztereinek értékei az aktuális utasítás végrehajtása után. Az alkalmazott ARM processzor esetén az R0 – R15 regiszterek értékei.

Processzor státusz
: A processzor állapotát leíró státuszregiszter értéke. ARM processzor esetén ez a CPSR (Current Program Status Register) értéke.

Utasítás mnemonic
: Az utasítás műveleti kódja alapján a Giano által visszafejtett mnemonic.

Utasítás argumentumok
: Az adott utasítás argumentumai (szintén a műveleti kód alapján).

Regiszter hozzáférések
: Az utasítás által végrehajtott regiszterműveletek, azaz hogy az utasítás melyik regisztereket olvasta és melyikeket írta a végrehajtása során.

Memória hozzáférések
: Az utasítás által végrehajtott memória-hozzáférések. Megadja, hogy az utasítás hatására melyik memóriacímeket olvasta (load) vagy melyeket írta (store) a processzor.

A rögzített adatok két szöveges fájlban érhetőek el, amelyek pontos formátumának ismertetésétől itt eltekintek.

Amint a fenti felsorolásból látható, a Giano valóban naplózza a program utasításainak szinte összes hatását és következményét. Amint a későbbiekben látható lesz, ezen információk megfelelő feldolgozása után a program futásával kapcsolatos legtöbb kérdésünkre választ kaphatunk.

4.2. Callgrind formátum

Miután a Callgrind eszközkészlet felépítését és használati lehetőségeit már az 1.3.4. fejezetben ismertettem, ebben a fejezetben csak a költségek rögzítéséhez használt fájlformátumot fogom röviden bemutatni.

A Callgrind által használt formátum tartalmazza a program függvényei közötti hívási kapcsolatokra vonatkozó információkat, a feltételes és feltétel nélküli ugrások adatait valamint az utasításokhoz rendelt költségeket is[18]. A Callgrind által generált adatfájl egyszerű szöveges formátumú, így mind az értelmezése, mind pedig az előállítása igen könnyű.

A Callgrind fájlokat felépítésük szerint lényegében két részre oszthatjuk, amelyek a fejléc és a tényleges futási adatokat tartalmazó törzs. A fejlécben kulcs – érték párok találhatóak, amelyek a fájl törzsében rögzített adatokra vonatkozó információkat tárolják. Ahogy később látni fogjuk, a fejlécben elsődlegesen a törzsben rögzített adatok jelentését kell megadni. Ezek mellett a fejléc a vizsgált programra vonatkozóan is tárolhat információkat.

A fejlécben két kulcs megadása lényeges. Ezek a `positions` és az `events`. A `positions` kulcshoz tartozó érték mezőben a költségsorokban megjelenő alpozíció meghatározásokat kell felsorolnunk. Ilyen alpozíció lehet a forráskód sor (`line`), vagy az adott utasítás memória címe (`instr`). Lehetséges több pozíció megadás kombinációja is (például utasítás cím és forráskód sor együttesen). Ezek az alpozíciók adják meg, hogy az adott költségsorban rögzített eseményt a programnak pontosan melyik utasítása (vagy forrás sora) okozta.

A költségek (ebben a kontextusban események) az `events` kulcs után definiálhatók. Költség lehet például az utasítás végrehajtásához szükséges processzor ciklusok száma vagy akár a különféle gyorsítótár műveletek is. Ahogy az alpozíciók esetén, itt szintén több eseményt is megadhatunk. A 4.3. listán egy valódi Callgrind fájl részlete látható. Megfigyelhető, hogy

4.3. lista. *Callgrind fájl részlet*

```
positions: instr line
events: Ir
summary: 691121
ob=hello.elf
...
fl=djpeg.c
fn=null'_start'main
0x92c 348 1
0x930 366 1
0x934 348 1
0x938 366 1
0x93c 366 1
```

a `positions` kulccsal a költségsorokhoz két alpozíciót rendelünk, amelyek az utasítás címe és a forráskód sor száma. Ahogy az `events` sorból megállapítható a rögzített esemény – azaz a költség – ebben az esetben az utasítás felhozatal (`Ir`).

A költségsorok a fejlécben megadott alpozíció értékeket és a pozícióhoz tartozó eseményeket tartalmazzák. Több alpozíció vagy esemény esetén az adatok a fejlécben megadott sorrendben szerepelnek. Az egymás utáni mezőket szóközök választják el.

A költségsorokat fájlokhoz (praktikusan forrásfájlok) és azokon belül függvényekhez rendelhetjük. Ennek megadására szolgálnak az `fl=` és a `fn=` sorok. Az `fl=` parancs a forrásfájl, míg az `fn=` a függvény nevének megadására használható. A beállított fájl- és függvénynév a következő `fl=` és `fn=` parancsokig minden költségsorra érvényes.

Hogy a programban fennálló kapcsolatok is rögzíthetőek legyenek, a Callgrind formátum speciális költségsorokat biztosít az ugró (feltételes vagy feltétel nélküli) és a szubrutin hívó utasítások reprezentálásához. A szubrutin hívások leírását a 4.4. listán látható példával ismertetem. Az első költségsor a hívó utasítást jelöli. Ezután következnek a meghívott

4.4. lista. *Callgrind függvényhívás részlet*

```
0x2b8 150 1
cfl=djpeg.c
cfn=null'_start'main
calls=1 0x92c 0
* * 14990
```

függvényt azonosító (`cfl=` és `cfn=`) sorok, amelyek a korábban bemutatott `fl=` és a `fn=` sorokhoz hasonlóan a meghívott függvényt tartalmazó forrásfájlt és a függvény nevét adják meg. A Callgrind (ahogyan 1.3.4. fejezetben láthattuk) a függvényhívások esetén eltárolja, hogy egy adott függvényhívás a program futása során hány alkalommal hajtódott végre. Ezt az információt tárolja a példán látható `calls=` sor első mezője. A további mezők a meghívott alpozíció azonosítását végzik, az általános költségsoroknál leírtaknak megfelelő módon. A következő sorban a `calls=` parancshoz tartozó költségsor látható, amely a meghívott függvény összesített költségét rögzíti. A csillag karakterek az alpozíciót megadó mezőkben azt jelzik, hogy az adott költségsor pozíciója azonos az előző költségsor pozíciójával.

A feltételes és feltétel nélküli ugró utasításokat bemutató két részlet a 4.5. listán látható. A feltétel nélküli vezérlésátadásokat a `jump` kulcsszó reprezentálja. Az ugrások leírásának

4.5. lista. *Callgrind ugró utasítások példa*

```
0x8ab0 872 1
jump=1 0x7a78 872
* *
...
0x7a9c 749 9
jcnd=5/9 0x7aa0 749
* *
```

szintaxisa nagymértékben hasonlít a szubrutin hívásokéra. Az első költségsor magának az ugrást kiváltó utasításnak a költségét rögzíti. A `jump` kulcsszó után következik a végrehajtott ugrások száma, majd az ugrás céljának megadása a szokásos pozíciómegadást követve. A szubrutin hívó utasításoktól eltérően itt a `jump` kulcsszó után szereplő költségsor nulla költségeket tartalmaz, hiszen a végrehajtás egy másik ponton folytatódik és nem tér vissza az adott pozícióba.

A feltételes ugrások rögzítéséhez a `jump` helyett a `jcnd` kulcsszó használatos. A Callgrind a feltételes ugrások esetén nem csak a bekövetkezett ugrások számát rögzíti, de azt is, hogy a program összesen hány alkalommal érte el az adott utasítást (a feltétel teljesülésétől függetlenül). Ennek megfelelően a `jcnd` parancs első mezője a tényleges ugrások számát és az ugró utasítás összes végrehajtásainak a számát rögzíti perjellel elválasztva. Amint a 4.5. példán látható, a program kilenc alkalommal futott rá a feltételes ugró utasításra, amely alkalmak közül tényleges vezérlésátadás ötször történt.

Annak érdekében, hogy a Callgrind fájlok megjelenítésekor a forráskód sorszámokat és az utasítás címeket össze lehessen kapcsolni a tényleges forráskóddal illetve assembly utasításlistával, a megjelenítőnek meg kell adni az ehhez szükséges információkat tartalmazó ELF fájlt. Ehhez a 4.3. példán megfigyelhető `ob=` parancs használható.

A Callgrind fájl fejlécében található `summary` kulcshoz tartozó érték megadja, hogy a vizsgált program futása összesítve milyen költségeket jelentett. Több eseményszám rögzítése esetén itt az egyes események egymástól független összesítése található meg. Az összesítés a Callgrind fájlok legvégén ismét megjelenik, ott a `totals` kulcsszóval.

A Callgrind formátum lehetőséget biztosít az adatfájlok „tömörítésére”, hogy a fájlok mérete még nagyobb programok vizsgálata esetén is kézben tartható legyen. Lényegében kétféle tömörítési mód adott. A függvények és forrásfájlok (általánosabban a sztringek) tömörítéséhez az úgynevezett név tömörítés használható. Ennek lényege, hogy a Callgrind fájlban többször előforduló – bizonyos esetekben igen hosszú – sztringekhez első megjelenésükkor egy egyedi azonosítót (pozitív egész szám) rendelünk. A sztringek további előfordulásakor már elegendő csak az azonosító rögzítése.

Az alpozíció tömörítés, ahogyan az elnevezése is mutatja, a költségsorokban megjelenő alpozíció meghatározások rövidített leírását teszi lehetővé. A programok nagyrészt lineáris végrehajtása (eltekintve a különféle vezérlésátadó utasításoktól) miatt az egymás utáni költségsorok pozíciói csak kis mértékben térnek el egymástól. Ezt használja ki az alpozíció tömörítés, amely lehetővé teszi, hogy a teljes pozíció meghatározás helyett csupán az aktuális és az előző pozíció különbségét rögzítsük. A 4.6. listán egy valódi Callgrind fájl részletén figyelhető meg az alpozíció tömörítés. Amint látható az első költségsorban mind-

4.6. lista. *Callgrind alpozíció tömörítés példa*

```
0xaa10 379 1
+3 * 1
+2 * 1
+6 +1 1
```

két alpozíció tömörítés nélkül szerepel. A további sorok azonban már csak a különbségeket rögzítik. A csillag karakterek a nulla különbséget jelölik, azaz hogy az aktuális sor alpozíciója megegyezik az előző soréval. Belátható, hogy ez a két tömörítési mód lényegesen csökkentheti a Callgrind fájlok méterét.

A Callgrind formátumban előállított hívási gráfokhoz a KCachegrind megjelenítő igen jól használható, más jellegű eredmények (például gráfok) bemutatásához azonban eltérő segédprogramok szükségesek. A következő fejezetekben ezek rövid bemutatása olvasható.

4.3. Gráf megjelenítés

A feladat megoldása során több esetben is a kinyert információkat egy gráf reprezentálja (hívási gráfok, adatfolyam gráfok). Ezért érdemes megvizsgálni, hogy milyen lehetőségek érhetőek el gráfok megjelenítésére. A sokféle lehetőség között felmerülhet valamilyen saját fejlesztésű eszköz létrehozása, azonban mérlegelve az ehhez szükséges erőfeszítéseket és az elérhető kész megoldásokat, ez az utat legfeljebb nagyon speciális igények esetén érdemes választani. Az elérhető kész megoldások közül kettőt mutatok be a továbbiakban.

Az első – és a legkézenfekvőbb – lehetőség a Graphviz[19] használata. A Graphviz egy ingyenesen elérhető, nyílt forráskódú eszközkészlet gráfok megjelenítésére, amelyet a legkülönfélébb tudományos és mérnöki területeken is használnak.

A Graphviz működésének lényege, hogy az ábrázolni kívánt gráfot egy egyszerű szöveges nyelven (DOT) leírjuk, és ezen leírás alapján a Graphviz elkészíti a gráf grafikus reprezentációját. A kimeneti ábra formátuma lehet egyszerű kép (bmp, gif, jpg, png, ...), de ha az eredményt egy dokumentumba szeretnénk beilleszteni PDF vagy PostScript formátum is választható. A DOT formátum egyszerűsége miatt a gráfok leírása könnyen előállítható akár kézzel, akár valamilyen program kimeneteként. A formátum részletes leírása a 4.3.1. alfejezetben olvasható.

Mivel a különböző alkalmazási területeken eltérő típusú gráfok megjelenítése a feladat, a Graphviz a kész ábrán a csomópontok és élek elrendezésére többféle algoritmust is biztosít. Így minden gráftípushoz kiválasztható az optimális elrendezést létrehozó algoritmus. Irányított élek és hierarchikus jellegű gráfok esetén a általában a `dot` parancs eredményezi a legjobban használható ábrákat. A `neato` és `fdp` parancsok az optimális elrendezéshez az éleket rugalmas kapcsolatokkal modellezik, és a teljes rendszer energia összegét minimalizálják. Ezeken túl egyéb speciális elrendezések (például körkörös) is elérhetőek.

A Graphviz további előnye, hogy a gráfokat reprezentáló csomópontok és élek grafikus megjelenítési módja igen nagymértékben befolyásolható. Megadható például a csomópontok alakja, színe, vonalvastagsága, a feliratok mérete és elhelyezése, valamint az élek vastagsága, színe és stílusa. Ezeket a tulajdonságokat az egyes elemekhez külön-külön vagy globálisan is megadhatjuk. A számtalan beállítási lehetőség jobb megértését nagymértékben segíti a Graphviz honlapján elérhető minta gráfok tanulmányozása, amelyekhez az azokat leíró DOT fájl is rendelkezésre áll.

A Graphviz által generált ábrák interaktívvá tételét teszi lehetővé az XDot projekt[20], amellyel a gráf csomópontjait vagy éleit tehetjük kattintásra érzékennyé. Így könnyen megvalósítható lehet például az egyes élekhez vagy csomópontokhoz kapcsolt extra információk megjelenítése az adott elemre való kattintás hatására.

A felsorolt előnyök mellett a Graphviz egy hátránya is említést igényel. A gráf elemeinek automatikus elrendezése csak bonyolult és indirekt módon befolyásolható. Ez nem jelent problémát ha a felkínált megjelenés kielégítő, azonban nagy méretű gráfok esetén az automatikus elrendezés nem mindig eredményez jól átlátható ábrákat. Ilyen esetekben – a gráf előállításának módjából adódóan – csak a gráf leírásának módosításával próbálkozhatunk, ami nem mindig vezet eredményre.

A Graphviz mellett egyéb gráf megjelenítő alkalmazások is említést érdemelnek. Ezek közül egy a yED gráf szerkesztő[21], amely gráfok és egyéb diagramok (például folyamatábrák és UML diagramok) előállítására használható. A yED a Graphviz-hez hasonlóan szintén ingyenes, azonban a forráskódja nem elérhető.

A yED támogatja a gráfok kézzel történő összeállítását, a csomópontok és az élek grafikusan adhatók meg. A gráf elemei tetszőlegesen mozgathatók, de a Graphviz-ben meglévőhöz hasonló algoritmusok is rendelkezésre állnak az automatikus elrendezésre. Természetesen a gráf elemeinek megjelenése (stílus, szín, méret) a yED esetén is tetszőlegesen beállítható.

A yED a gráfok kézi megadása mellett többféle gráf leíró formátumot támogat a csomópontok és élek beviteléhez és tárolásához. Ezek között a formátumok között találhatóak egyszerű szöveges megadást támogató és XML alapú leírások is. A Graphviz DOT formátuma nem támogatott, így a két eszköz nem közvetlenül kompatibilis. A yED által kezelt GML (Graph Modelling Language) formátum azonban a DOT-hoz hasonlóan egyszerű szöveges leírás, amely szintén könnyedén generálható akár kézzel, akár valamilyen program kimeneteként. A kimeneti formátum egyszerű módosításával a két megjelenítő eszköz számára a gráf leírása tehát hasonlóan előállítható.

A yED a gráf bevitele és a csomópontok elrendezése után, a Graphviz-hez hasonlóan, a kimeneti képformátumok széles választékát nyújtja. Itt is megtalálhatóak az alapvető képformátumok mellett a PDF és a PostScript is.

A Graphviz döntő előnye a yED-el (és más hasonló programokkal) szemben az, hogy amíg a Graphviz az XDot megjelenítő által beépíthető saját készítésű alkalmazásokba, addig az önálló alkalmazásként elérhető yED esetén ez nem kivitelezhető, így az interaktív jellegű gráfok létrehozása nem lehetséges.

A fent összefoglalt szempontokat figyelembe véve a gráfok ábrázolásához a Graphviz eszközkészletét használtam a feladat megoldása során. Ennek megfelelően a következőkben a DOT formátum rövid bemutatása olvasható.

4.3.1. DOT formátum

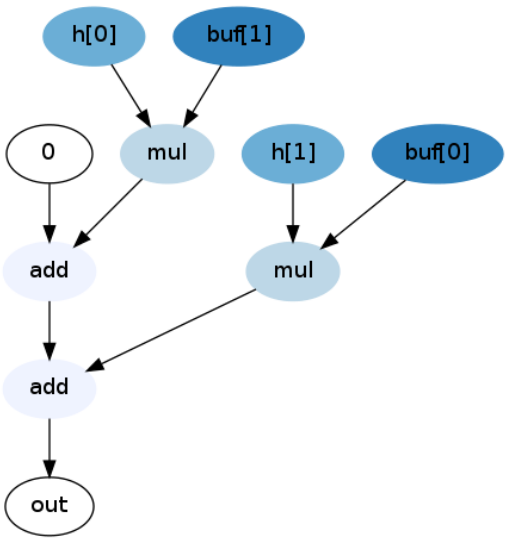
A Graphviz által a gráfok leírására használt DOT nyelvet a 4.7. listán látható példa alapján mutatom be, amely a 4.2. ábrán látható kimenetet eredményezi.

A DOT fájlok lényegében a csomópontok és az élek felsorolását tartalmazzák. A csomópontok megadásánál elsőként meg kell adni a csomópont azonosítóját, majd szögletes zárójelek között a beállítani kívánt tulajdonság – érték párokat vesszővel elválasztva. A példán látható csomópont definíciók esetében a megjelenített felirat, a stílus és a kitöltési szín kerül beállításra.

Az irányított élek esetén a `->` jelet használva kell megadni az összekötni kívánt két csomópont azonosítóját, majd szögletes zárójelek között a tulajdonságok is beállíthatók. Az élekre vonatkozóan is kiválasztható például a stílus, szín vagy a felirat. A példán látható `weight` paraméterrel az él „súlya” állítható be, amely a gráf elemeinek elrendezését befolyásolja úgy, hogy a nagyobb súlyú élek minél függőlegesebbek és rövidebbek legyenek.

4.7. lista. *Graphviz DOT példa*

```
digraph ddfg{
        null[label="0"];

        add1[label="add", style=filled, color="/blues5/1"];
        add2[label="add", style=filled, color="/blues5/1"];

        mul1[label="mul", style=filled, color="/blues5/2"];
        mul2[label="mul", style=filled, color="/blues5/2"];

        out[label="out"];

        h0[label="h[0]", style=filled, color="/blues5/3"];
        h1[label="h[1]", style=filled, color="/blues5/3"];

        d0[label="buf[0]", style=filled, color="/blues5/4"];
        d1[label="buf[1]", style=filled, color="/blues5/4"];


        null -> add1[weight=10];
        add1 -> add2[weight=10];
        add2 -> out[weight=10];

        mul1 -> add1;
        mul2 -> add2;

        h0 -> mul1;
        d1 -> mul1;

        h1 -> mul2;
        d0 -> mul2;
}
```



4.2. ábra. *Graphviz DOT példa kimenet*

5. fejezet

Algoritmusok analízisének megvalósítása

Ebben a fejezetben az algoritmusok vizsgálatához létrehozott eszközt mutatom be részletesen. Elsőként az analízis eszköz követelmény specifikációja olvasható, majd a megvalósításhoz használt eszközöket ismertetem. Az eszköz magas szintű leírása után a modulok és osztályok felépítésének, működésének és kapcsolatainak részletesebb elemzése következik.

5.1. Az analízis eszköz követelmény specifikációja

Az elkészítendő analízis eszköznek a következő funkcionális követelményeknek kell megfelelnie:

1. A vizsgált program Giano alatt történő szimulációjáról naplózott információk értelmezése, feldolgozása, és az alapján a további követelményekben felsorolt kimenetek előállítása.

2. A vizsgált program fordításakor keletkező ELF fájl értelmezése, és a vizsgálathoz szükséges információk kinyerése.

3. Az analízis eszköznek a bemeneti adatok alapján elő kell állítania a vizsgált program futási adatait és hívási gráfját tartalmazó Callgrind formátumú kimenetet, amely a KCachegrind használatával jeleníthető meg.

4. Az analízis eszköznek a bemeneti információkból ki kell nyernie a vizsgált program dinamikus adatfolyam gráfját, és az alapján a következő pontokban leírt további kimeneteket kell előállítania.

5. A dinamikus adatfolyam információkat felhasználva az eszköznek meg kell határoznia a vizsgált program részei (például forrásfájlok és függvények) közötti adatáramlás tulajdonságait (mennyiség, időbeli és „térbeli” eloszlás).

6. A program részei közötti adatáramlást grafikusan, jól átlátható és értelmezhető módon kell megjeleníteni.

A megvalósításra vonatkozóan a következő – szinte általános – követelményeket határoztam meg:

1. Az elkészített eszköz felépítésének modulárisnak kell lennie, hogy megkönnyítse a későbbi bővítést, módosítást.

2. Szintén a további fejlesztést megkönnyítendő a forráskódnak jól dokumentáltnak kell lennie, lehetőség szerint valamilyen dokumentáció generáló eszközt (például Doxygen) használva.

3. A fejlesztési folyamat jobb átláthatósága érdekében verziókezelés használata.

5.2. A megvalósítás eszközei

Ebben a fejezetben a fent felsorolt követelményeknek megfelelő analízis alkalmazás elkészítéséhez használt eszközöket mutatom be érintőlegesen.

5.2.1. Programozási környezet

Az analízis eszköz implementálásához használt programozási nyelv kiválasztásánál figyelembe kellett vennem a megfogalmazott követelményeket, különös tekintettel a moduláris felépítésre és a továbbfejleszthetőségre vonatkozóan. Ennek megfelelően az optimális egy objektumorientált programozást támogató nyelv használata, hiszen az objektumorientált megoldások biztosítják a nagymértékű modularitást és a kód újrahasznosíthatóságát. Az ilyen nyelvek közül a C++ és a Java a legnépszerűbb és a legáltalánosabban elterjedt. Ezzel szemben a feladat megoldásához a Python[22] nyelvet választottam, a következőkben felsorolt indokok alapján.

- A Python is támogatja a moduláris felépítést és az objektumorientált megoldások nagy részét a C++-hoz vagy a Java-hoz hasonlóan.

- Mivel a Python nagyon magas szintű absztrakciót tesz lehetővé, a programok fejlesztése kisebb mértékű energia befektetésével és viszonylag gyorsan végezhető. (Az erre a pontra vonatkozó személyes tapasztalataim is befolyásolták a választást.)

- Szintén a gyors fejlesztést teszik lehetővé a Python széleskörű, jól dokumentált és jól használható szabványos könyvtárai.

- Az elkészített kód jól dokumentálható (Doxygen jellegű dokumentációs sztringek) és jól olvasható.

A Python felsorolt pozitív tulajdonságai mellett természetesen néhány – ezekkel összefüggő – hátrányos tényezőt is figyelembe kell venni. Mivel a Python nyelv interpretált, az ezen a nyelven írt programok futtatásához Python értelmező szükséges, és a futási sebesség is elmarad például egy tiszta C nyelvű megvalósítástól. Ennek a hátránynak a csökkentése érdekében a Python interpreter a futtatott kód köztes reprezentációjaként először egy

bájt-kódot készít, majd ezt értelmezi és hajtja végre. Így a végrehajtási sebesség az elfogadható tartományban tartható. Mivel az elkészített eszköz alapvetően kutatási célokat szolgál és semmilyen szempontból nem tekinthető kész terméknek, a gyors és hatékony fejlesztés lehetősége ellensúlyozza a valamivel kisebb futási teljesítményt.

A Python egy további hátrányaként vehető számításba, hogy Java vagy C++ nyelvekhez képest kisebb az elterjedtsége, ami a további fejlesztést nehezítheti némileg.

Fontos kitérni a Python dinamikus típuskezelésére, amely a C++ vagy Java nyelvektől jelentősen eltérő. Amíg a statikus típusosság esetén a változókhoz rendelhetünk típusokat, a Python csak az értékekhez rendel típusokat, a változók tetszőleges típusú értékeket tárolhatnak. Ennek a megközelítésnek az előnye, hogy a változókat a típus megadása nélkül használhatjuk, ami rugalmasabb fejlesztést tesz lehetővé. A dinamikus típusosság hátránya viszont, hogy mivel a programozó nincs rákényszerítve a típusok explicit megadására, a hibázás lehetősége nagyobb. Ezt ellensúlyozzák a futási időben alkalmazott ellenőrzések, amelyek hibás típusmegadás esetén kivételt generálnak.

Fontos megemlíteni, hogy a Python esetében – más nyelvekkel ellentétben – minden objektum (még a függvények is). Ennek több következménye is van, amelyek közül egyet szeretnék kiemelni: A Python változói minden esetben objektumokra mutató referenciákat tárolnak. Ezt a tulajdonságot például paraméterátadások esetén kiemelten figyelembe kell venni. A félreértések elkerülése érdekében a következőkben az „egy változó egy objektumot tárol” kifejezést minden esetben úgy kell értelmezni, hogy a változó az objektumra mutató referenciát tartalmazza.

Dokumentáció generálás

Az elkészített eszköz fejlesztése közben a forráskód alapján történő dokumentáció generáláshoz az EpyDoc[23] segédeszközt használtam. Ez az eszköz a Python nyelvű forráskódokban elhelyezett Doxygen stílusú dokumentációs sztringek alapján automatikusan előállítja a modulok, osztályok és metódusok jól formázott és böngészhető dokumentációját. Lényeges kiemelni, hogy az EpyDoc lehetőséget biztosít a függvényparaméterek és visszatérési értékek típusainak megadására is, ami különösen előnyös a Python dinamikus típusosságát figyelembe véve.

Profiling

Ahogyan az 1. fejezetben már bemutattam, a profiling a szoftverfejlesztés egy fontos eszköze. Mivel az elkészült analízis eszköznek jelentős mennyiségű információ feldolgozása a feladata, a futási idő elfogadható szinten tartása fontos tényező. Ennek megfelelően, a fejlesztés során a Python beépített profiling lehetőségeit kihasználva, a megfelelő részek optimalizálásával a feldolgozási időt jelentősen tudtam csökkenteni.

5.2.2. Verziókezelés

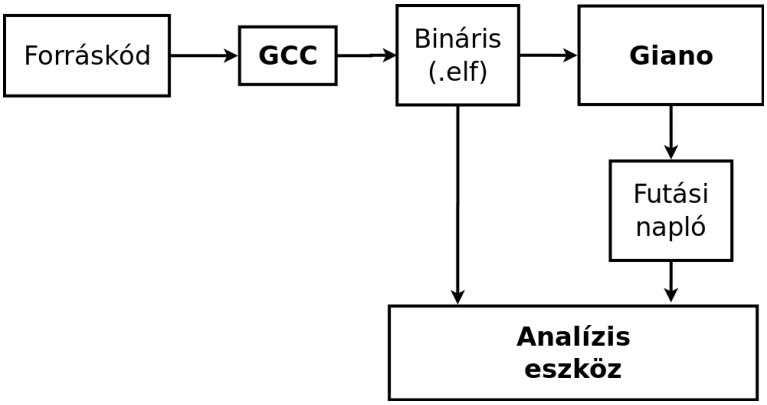
Az analízis eszköz fejlesztése során a fejlesztés lépéseinek követésére a Git[25] verziókezelő eszközt használtam. A Git egy nyílt forráskódú elosztott verziókezelő rendszer, amely széles

körben használt az egészen kis alkalmazásoktól a nagyon komplex rendszerekig. Az elosztott működésből adódóan a Git repository létrehozásához és használatához nincsen szükség központi szerver üzemeltetésére, ami jelentősen megkönnyíti a használatát.

5.3. Magas szintű felépítés

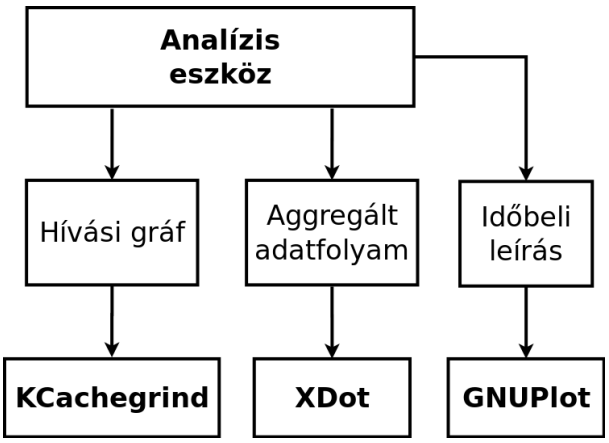
Ebben a fejezetben a megvalósított analízis program magas szintű felépítését mutatom be. A felépítés mellett az eszköz külső kapcsolatait is ismertetem, azaz hogy milyen bemeneteket fogad, és milyen kimeneteket szolgáltat. A kapcsolódó külső eszközöket is megemlítem. Ezek után a program működésének belső lépéseit vázolom szintén magas szinten.

Az 5.1. ábrán az analízishez szükséges bemenetek előállítása látható a vizsgálni kívánt program forráskódjától kezdődően. Megfigyelhető, hogy az első lépés a vizsgálat tárgyát



5.1. ábra. *Az analízis eszköz bemenetei*

képező program lefordítása a GCC fordítóval. Ezen lépés során a forráskódból létrejön a futtatható gépi kódot és sok egyéb fontos információt tartalmazó ELF (Executable and Linkable Format) fájl. A futtatható gépi kód ezután bekerül a Giano szimulátorba, amely lefuttatja azt, és kimenetként előállítja a futás minden részletre kiterjedő naplóját. Az analízis eszköz ezek után bemenetként fogadja ezt naplót, valamint az ELF fájlt, amelyből a teljes analízishez szükséges statikus jellegű információkat nyeri ki.

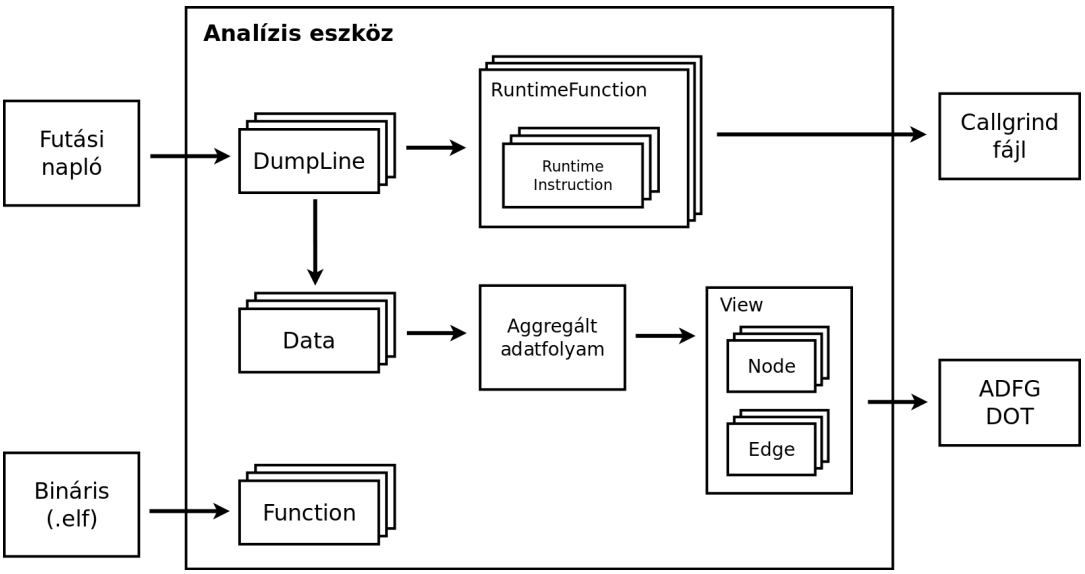


5.2. ábra. *Az analízis eszköz kimenetei*

Az analízis program kimenetei és a kapcsolódó megjelenítők az 5.2. ábrán figyelhetők meg. A program függvényei közötti hívási kapcsolatokat leíró Callgrind formátumú kimeneti fájlt a már bemutatott KCachegrind jeleníti meg. A függvények közötti adatáramlásokat ábrázoló aggregált dinamikus adatfolyam gráf DOT (l. 4.3. fejezet) formátumban jön létre, amelyet az XDot megjelenítő alakít interaktívan használható ábrázolássá. A felhasználói bemenet (egér kattintás egy csomóponton vagy élen) hatására az analízis eszköz előállítja az adott elemre vonatkozó időbeli információkat, majd átadja azokat a GNUPlot megjelenítő számára, amely grafikonon ábrázolja az adatokat.

5.3.1. A feldolgozás lépései

Ebben az alfejezetben az analízis eszköz működésének lépéseit vázolom fel, hogy a későbbiekben – a részletek bemutatásakor – az egyes részegységek viszonyának megértése könnyebb legyen. A felépítést az 5.3. ábra alapján fogom bemutatni. Az ábrán a vizsgált programot jellemző információ halmaz különböző reprezentációit láthatjuk. Fontos megjegyezni, hogy az ábra nem mutatja többek között az objektumok közötti kapcsolatokat, valamint a kevésbé lényeges objektumokat sem ábrázoltam.



5.3. ábra. *Az analízis eszköz belső működésének áttekintése*

A Giano napló feldolgozásának első lépése a napló fájlok beolvasása. A beolvasott sorokat, azaz végrehajtott utasításokat, az eszköz `DumpLine` objektumokkal reprezentálja. Az egyes objektumok létrehozásához be kell olvasni és fel kell dolgozni a napló fájlok mezőit, majd az információk alapján példányosítani kell a végrehajtásnak megfelelő `DumpLine` objektumokat. A `DumpLine` objektumok lényege, hogy egységesen tárolják a végrehajtott utasítások minden aspektusát, amellyel a későbbi feldolgozás nagyban megkönnyíthető.

A Callgrind kimenet előállításához az analízis eszköz a `DumpLine` objektumok alapján `RuntimeFunction` és `RuntimeInstruction` objektumokat példányosít, amelyek lényegében a Callgrind formátum egy belső reprezentációját képezik. A `RuntimeFunction` példányok a program függvényeinek feleltethetők meg kontextus-érzékeny módon (l. 1. fejezet), azaz kü-

lönböző `RuntimeFunction` tartozik egy adott függvényhez ha az különböző kontextusokból hívódott meg a program futása során. Ennek megfelelően elmondható, hogy a kimenetként előálló hívási gráf is kontextus érzékeny lesz.

A `RuntimeInstruction` objektumok a `RuntimeFunction` függvényekben végrehajtott utasításokat modellezik. A Callgrind formátumnak megfelelően a `RuntimeInstruction` speciális változatai használhatók ugró és szubrutinhívó utasítások esetén. Ezek az objektumok lényegében a Callgrind költségsorainak felelnek meg, az azok létrehozásához szükséges információ található bennük.

Miután a `RuntimeFunction` és `RuntimeInstruction` objektumok létrejöttek, a következő lépés a tényleges hívási gráf kimenet előállítása.

A dinamikus adatfolyam gráf felvétele szintén a `DumpLine` objektumok alapján történik. A vizsgált program futása során előálló adatokat `Data` objektumok modellezik. Miután a gráf élei az adatok létrehozásához felhasznált adatokat mutatják, a `Data` objektumok referenciákat tartalmaznak a forrás-adatokat reprezentáló `Data` objektumokra. Ehhez a lépéshez igen nagy segítséget nyújt, hogy a `DumpLine`-ok, azaz a Giano napló bejegyzések, a regiszter- és memóriaműveleteket is rögzítik.

A következő lépés az adatfolyam gráf aggregálása, azaz az adatfolyam gráf csomópontjainak és éleinek összesítése. Amint látható, az 5.3. ábrán az aggregált adatfolyam belső felépítését nem tüntettem fel, mivel az ábrázolás az összetett felépítés miatt nehézkes lett volna. Az aggregálás lényege, hogy a rögzített adatfolyam elemeit hierarchikusan (modulok, függvények és utasítások szerint) egymásba ágyazott csomópont és él objektumok írják le.

Az aggregált adatfolyam (ADFG – Aggregated Data Flow Graph) alapján előállított nézetek lényegében csak a kimenet előállítását egyszerűsítik. Egy nézet az aggregált adatfolyam egy adott hierarchia szintjén található csomópontokat és a közöttük haladó éleket tartalmazza, nem-hierarchikus (flat) formában. A gráf elemeiből ezután könnyedén előállítható a kimeneti reprezentáció DOT vagy GML formátumban.

Az 5.3. ábrán az eddigiek mellett a vizsgált programhoz tartozó ELF fájl egy fontos felhasználása is látható. A program függvényeinek nevei és memóriabeli elhelyezkedésük `Function` objektumokban kerülnek rögzítésre. Ezen információk fő felhasználása a `RuntimeFunction` objektumok létrehozásánál található.

A fentiek az analízis program működésének egy igen felszínes áttekintését mutatják, azonban az információ áramlása és a feldolgozási lépések láthatók.

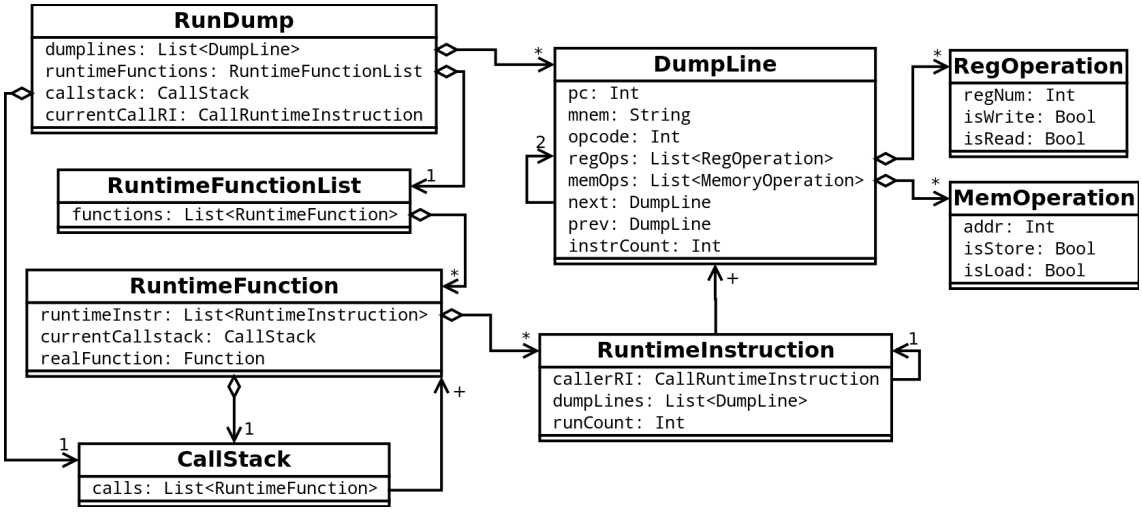
A továbbiakban az analízis program moduljainak és osztályainak részletes felépítését és működését, valamint az osztályok kapcsolatait fogom bemutatni.

5.4. A megvalósítás részletei

Ebben a fejezetben az analízis eszköz részleteit ismertetem. A program moduljait és osztályait azok logikai sorrendje szerint fogom bemutatni, az osztályok közötti kapcsolatok figyelembevételével. A program teljes forráskódja a mellékletek között található, az osztályok és függvények tényleges megvalósítása ott megtekinthető.

5.4.1. Giano napló fájlok feldolgozása és Callgrind kimenet előállítása

Az 5.4. ábrán a Giano napló fájlok feldolgozásával és a Callgrind kimenet létrehozásával kapcsolatos osztályokat ábrázoló osztálydiagram látható. Fontos megjegyezni, hogy az ábrán nem tüntettem fel az osztályok metódusait, illetve attribútumaik közül is csak a bemutatás szempontjából fontosakat ábrázoltam.



5.4. ábra. *RunDump osztálydiagram*

RunDump

A `RunDump` osztály lényegében összefogja a Giano napló fájlok beolvasásához és a Callgrind jellegű futási információk kinyeréséhez szükséges funkciókat. A „főprogramban" praktikusan egyetlen RunDump példányt kell létrehozni, majd a megfelelő metódusokat meghívni.

A Giano naplófájlok által tartalmazott információk értelmezését nagyrészt a `DumpLine` objektumok végzik el. A `RunDump` feladata a napló fájl szétbontása az egyes utasítások végrehajtását leíró egységekre, üres `DumpLine` objektumok létrehozása, majd a napló fájl részek átadása. A végrehajtott utasítások számának nyilvántartása, és a `DumpLine` objektumok kétirányú láncolása is itt történik. A `RunDump` az adatokkal feltöltött `DumpLine` objektumokat egy listában tárolja a későbbi feldolgozáshoz.

A napló fájlok értelmezése után a következő lépés a hívási gráf információk feldolgozása. Ehhez a már kitöltött `DumpLine` objektumok használhatók. Ebben a lépésben a legfontosabb a függvényhívások és visszatérések érzékelése és követése. Az alkalmazott ARM processzor számára – más architektúrákkal szemben – a függvényhívásokat és visszatéréseket igen változatos utasításokkal írhatjuk le, amely tulajdonság bizonyos nehézséget jelenthet. A kihívást tovább nehezítheti ha a fordító ezeket a lehetőségeket ki is használja, például optimalizációs céllal. Az 5.1. és az 5.2. táblázatokban a `RunDump` által felismert szubrutinhívási és visszatérési utasítások láthatók. Fontos megjegyezni, hogy a felsorolt utasítások mellett azok feltételes alakja is előfordulhat. Mivel az ARM utasításkészlet felépítése igen egységes, az utasítások (feltételes vagy feltétel nélküli alakban) felismerése legegyszerűbben az utasításkódok alapján végezhető.

Kód	Leírás
`bl addr`	Branch with link, alapvető függvényhívási utasítás. Programszámláló mentése a Link regiszterbe, majd a végrehajtás folytatása a megadott címről.
`mov lr, pc` `mov pc, addr`	Lényegében a `bl` utasítás működését másolja. A `mov` utasítás által támogatott többféle címzési mód miatt lehet előnyös a használata.

5.1. táblázat. *ARM függvényhívási variációk*

Kód	Leírás
`mov pc, lr`	Programszámláló visszatöltése a Link Regiszterből. Legegyszerűbb visszatérési mód.
`pop {..., pc}`	Előző programszámláló érték visszatöltése a stack-ről az elmentett regiszterekkel együtt.
`bx lr`	Ugrás a Link regiszter által tárolt címre. Lényegében megegyezik az első lehetőséggel.

5.2. táblázat. *ARM visszatérési variációk*

A hívási gráf információk kinyeréséhez a `RunDump` a program működését rögzítő `DumpLine` objektumok feldolgozása közben követi a hívási és visszatérési utasításokat. Ezek alapján nyilvántartható az aktuális hívási sor, azaz hogy a program mely függvénye fut éppen, és milyen függvényeken keresztül jutottunk el az adott állapotba. Ez lényegében a hívási verem követését jelenti. Az aktuális verem állapotot a `callstack` tagváltozóban tárolt `CallStack` példány tárolja.

Hogy az előállított hívási gráf kontextus-érzékeny legyen, a `RuntimeFunction` objektumok nem csak az adott statikus függvény referenciáját tárolják, de a meghívásuk pillanatában aktuális `callstack` másolatát is. Így a különböző kontextusokból hívott azonos függvények megkülönböztethetők.

A `RunDump` minden feldolgozott `DumpLine` esetén létrehozza az adott utasításnak megfelelő típusú (l. később) `RuntimeInstruction` objektumot. Ha a program futása egy olyan függvénybe (kontextus-függően) került amely még nem szerepelt a futás során, egy új `RuntimeFunction` objektum is létrehozásra kerül, a függvény reprezentálására. A létrehozott objektumok közötti kapcsolatok megfelelő beállítása szintén igen fontos. A függvények számon tartják az őket alkotó utasításokat, és az utasítások is rendelkeznek referenciával az őket tartalmazó függvény felé.

Ahogy a későbbiekben részletesen is bemutatom, a Callgrind költségek terjesztéséhez szükséges, hogy minden `RuntimeInstruction` utasítás rendelkezzen az őt tartalmazó függvényt meghívó utasítás referenciájával. Ennek megvalósítását segíti a `RunDump` osztály `currentCallRI` tagváltozója, amely mindig az aktuálisan futó `RuntimeFunction` függvényt meghívó `RuntimeInstruction` példányra mutat. A létrehozott `RuntimeInstruction` objektumok `callerRI` tagváltozójának beállítása, és a `currentCallRI` aktualizálása függvényhívások és visszatérések esetén szintén a `RunDump` feladata.

DumpLine

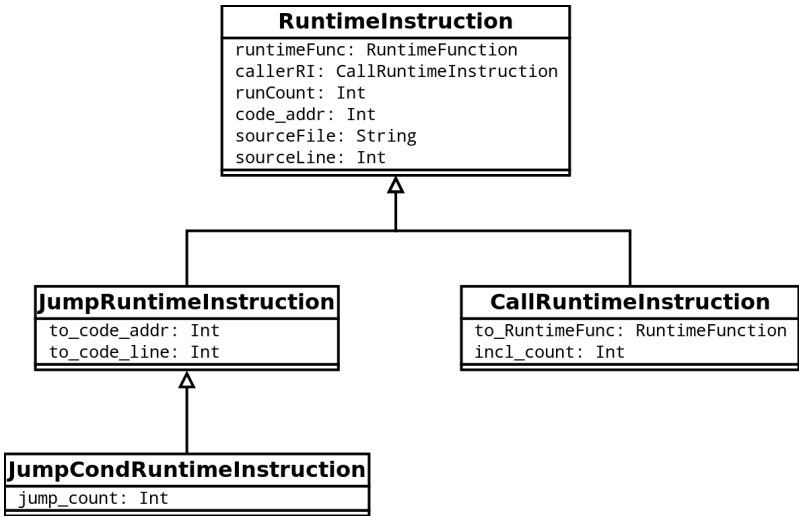
A `DumpLine` objektumok feladata a vizsgált program futása során végrehajtott utasítások jellemzőinek tárolása. A példányok létrehozásakor elsőként egy üres `DumpLine` keletkezik, majd a megfelelő metódusoknak átadott Giano napló részletek értelmezése után áll elő a tényleges `DumpLine`.

A példányok tárolják az utasítás memória címét (programszámláló érték), utasításkódját, az abból visszafejtett mnemonikot és argumentumokat. Elérhetőek továbbá az utasítás által végrehajtott memória- és regisztermű̈veleteket leíró `MemOperation` és `RegOperation` objektumok, valamint a végrehajtás idejét (a program futása során) megadó `instrCount` érték is. A `DumpLine` objektumok kétirányú láncolása (a `prev` és `next` tagok által) bizonyos feldolgozási lépéseket egyszerűsít nagy mértékben.

A `MemOperation` és `RegOperation` osztályok egy-egy memória- vagy regiszter műveletet írnak le. Regiszter művelet esetén a hozzáférés iránya (írás, olvasás vagy mindkettő) és az elért regiszter száma tárolódik. A memóriaműveletek reprezentációja mindössze annyiban különbözik, hogy a memóriarekesz címe kerül tárolásra, és egy-egy művelet vagy csak írás (store), vagy csak olvasás (load) lehet.

RuntimeInstruction

A Callgrind kimenet előállításában a `RuntimeInstruction` objektumok szerepe igen nagy, hiszen ezek a példányok lényegében közvetlenül a Callgrind költségsoroknak feleltethetők meg. A költségsorok különböző altípusainak megfelelően az általános `RuntimeInstruction` osztályhoz is definiáltam specializált alosztályokat. Az osztályok hierarchiáját az 5.5. ábrán figyelhetjük meg.



5.5. ábra. *RuntimeInstruction leszármazottak*

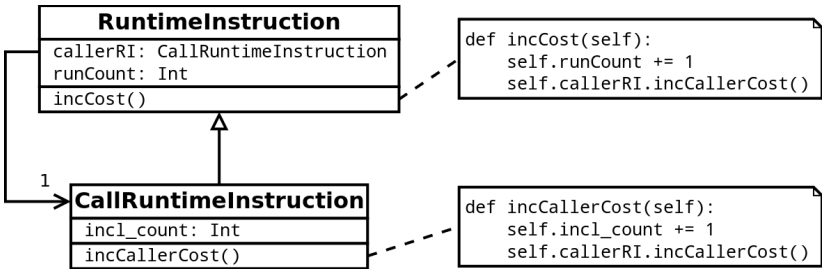
A normál (nem szubrutinhívó vagy ugró) utasítások leírására a `RuntimeInstruction` szolgál. Amint az ábrán látható, a példányok eltárolják az őket tartalmazó `RuntimeFunction` és az azt meghívó `RuntimeInstruction` referenciáját (`callerRI`). A `runCount` tagváltozó megfeleltethető a Callgrind által definiált költségnek. Ez jelen megvalósításban az adott

utasítás végrehajtásainak számát jelenti. A `code_addr`, `sourceFile` és `sourceLine` változók a költségsor pozíciójának megadásához szükséges adatokat tárolják, azaz az utasítás címét, az utasítást tartalmazó forrásfájl nevét és a forrás sor számát.

A függvényhívások leírásához a `CallRuntimeInstruction` utasítás használható. Itt megfigyelhető az ugrás célját megadó `to_RuntimeFunc` tagváltozó, amely a meghívott függvényt leíró `RuntimeFunction` példány referenciáját tárolja. Az `incl_count` tagváltozó az ebből az utasításból meghívott függvény összesített költségét tárolja.

A feltétel nélküli ugrásokat a `JumpRuntimeInstruction` osztály írja le. Az ugrás célja itt az utasítás címével és a forráskód sor számával van megadva. A feltétel nélküli ugrások esetén az ugrások számát az utasítás futási száma adja meg. A feltételes ugrásokat leíró `JumpCondRuntimeInstruction` esetén a tényleges ugrások számát (azaz hányszor teljesült a feltétel) a `jump_count` tagváltozó rögzíti.

Az utasítások költségeinek számítása az utasításokat és függvényeket reprezentáló objektumok közötti kapcsolatoknak köszönhetően igen egyszerűen megvalósítható. A költségek számításának alapja az utasítások továbbterjesztése a hívási láncban. Ennek megértéséhez fel kell idéznünk a Callgrind hívási költségsorait, amelyek a hívási utasítás futásainak száma mellett a meghívott függvényben keletkezett költségek összesített értékét is tartalmazzák. A hívási gráfban lefelé (a hívások irányában) haladva a költségek egyre több függvény között oszlanak szét. Ha egy adott függvényben keletkezik egy költség (végrehajtottunk egy utasítást), akkor azt a költséget a teljes hívási láncban vissza kell terjeszteni. Ez azt jelenti, hogy a költségnek meg kell jelennie a láncban található összes hívási költségsor összesített értékében. Ennek a megoldásnak az előnyös következménye, hogy a program végrehajtásának teljes költsége a legfelső szinten automatikusan kiadódik.



5.6. ábra. *RuntimeInstruction költség terjesztés megvalósítása*

A fent leírt működés megvalósítását az 5.6. ábra segítségével ismertetem. A működés kulcsa a minden `RuntimeInstruction` példányban megtalálható `callerRI` tagváltozó, amely az adott utasítást tartalmazó függvényt meghívó `CallRuntimeInstruction`-ra mutat. Egy utasítás végrehajtásának érzékelésekor a `RunDump` meghívja az adott utasítást reprezentáló `RuntimeInstruction` példány `incCost` metódusát. Ez a metódus, amint az 5.6. ábrán látható, elsőként megnöveli az adott utasítás `runCount` változóját, azaz a saját költségét. A költség továbbterjesztése érdekében ezután meghívja a `callerRI` tag `incCallerCost` metódusát, amely szintén megnöveli a saját összesített `incl_count` változóját, majd továbbítja a költséget a `incCallerCost` metódussal az őt tartalmazó függvényt meghívó `CallRuntimeInstruction` felé.

A legegyszerűbb megvalósítás érdekében szükség van egy kezdeti (a vizsgált programon kívüli) hívó utasításra, amely a program futása során előálló összes költséget rögzíti. A futás végén ez a hívás a program teljes költségét adja meg. Az ehhez szükséges `CallRuntimeInstruction` példányt a `RunDump` hozza létre a `DumpLine`-ok feldolgozása előtt. Ezt a példányt a `nullCallRI` tárolja, és a `currentCallRI` változó is kezdetben erre mutat. A program futása során az első utasítások `callerRI` referenciája így szintén a kezdeti hívó utasításra fog mutatni.

ELFHelper

A vizsgált program statikus információit tároló ELF fájl beolvasására, és a beolvasott információk tárolására az `ELFHelper` osztály szolgál. A vizsgálat szempontjából az ELF fájlban található szimbólum információk a legfontosabbak. Ilyen szimbólum információk többek között a függvények és globális változók kezdőcímei és hosszai, illetve a memória szekciók elhelyezkedése. Az `ELFHelper` metódusokat biztosít a kinyert információk több szempont szerinti kereséséhez. Függvények esetén például van lehetőség keresésre memóriacím alapján, de név szerint is.

Az `ELFHelper` konstruktorának átadott elérésű ELF fájl információinak feldolgozása azonnal megtörténik. Az információk kinyeréséhez az `ELFHelper` a megfelelő paraméterekkel meghívott `objdump` segédprogram kimenetét értelmezi, majd az információkat eltárolja. A gyors kereshetőség érdekében az információk egy része a Python által biztosított asszociatív tömb jellegű *dictionary* objektumokban kerül tárolásra.

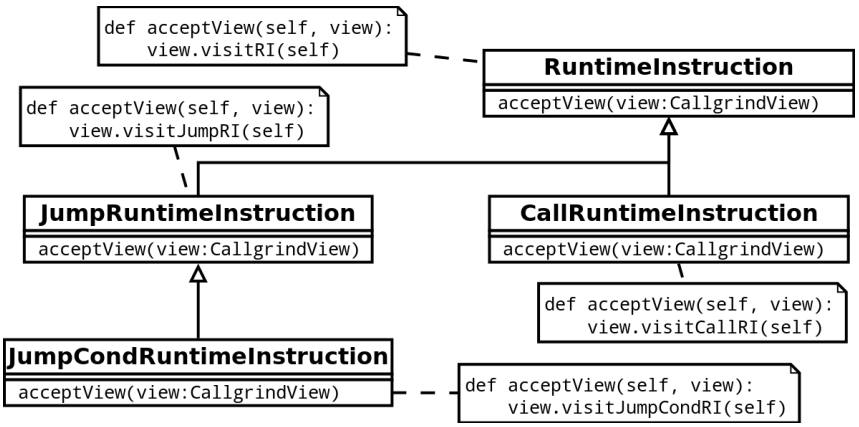
CallrgindView

Miután a hívási gráf információk előálltak a program vizsgálatával, a tényleges Callgrind formátumú kimenet előállítása a `CallgrindView` feladata. A példányosított `CallgrindView` a konstruktorban átadott `Rundump` objektum alapján készíti el a kimenetet.

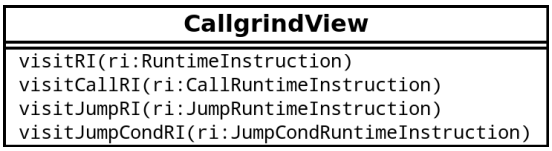
A Callgrind fájl fejlécében a 4.2. fejezetben bemutatottak szerint meg kell adni a pozíciók és események leírását, valamint a teljes költséget. Ezután következnek a program függvényei, azaz a `RuntimeFunction` objektumok. Minden függvény elején kiírásra kerül a forrásfájlt és a függvényt azonosító `fl=` és `fn=` sor, majd a függvényben található utasítások alapján képzett költségsorok. Mivel a különböző típusú `RuntimeInstruction` példányokhoz eltérő kimenetet kell generálni, ezeket meg kell tudni különböztetni. A típusok megkülönböztetésére alapvetően két megoldás használható. Az egyik lehetőség a Python `isinstance` parancsa, amellyel egy objektum típusát lehet meghatározni, a másik pedig a *Visitor* tervezési minta használata. A megvalósításhoz a *Visitor* tervezési mintát választottam, mivel az nagyobb rugalmasságot biztosíthat a későbbi módosítások számára.

A *Visitor* minta alapján létrehozott implementáció részletei az 5.7. és az 5.8. ábrákon figyelhetők meg.

A működés lényege, hogy a `CallgrindView` egy `RuntimeInstruction` kiírásához meghívja annak `acceptView` metódusát, saját referenciáját adva meg paraméterként. A metódus ezután az objektum típusának megfelelő `visit` metódust hívja vissza az átadott



5.7. ábra. *Visitor minta használata a CallgrindView osztályban I.*



5.8. ábra. *Visitor minta használata a CallgrindView osztályban II.*

`CallgrindView` példányon. A `CallgrindView visit` metódusai ezután már a pontos típusnak megfelelő kimenetet állíthatják elő a paraméterként kapott `RuntimeInstruction` objektumok alapján.

Fontos megjegyezni, hogy többek között a Python dinamikus típusossága miatt, a megvalósított kialakítás nem felel meg teljes mértékben a „klasszikus” *Visitor* mintának, azonban ez a lényegi működést nem befolyásolja.

5.4.2. Adatfolyam információk kinyerése

A következőkben az adatfolyam információk kinyerésének megvalósítását fogom bemutatni, amely megvalósítás több szempont szerint is igen hasonlatos a az 1.4.1. fejezetben látott Valgrind – Redux eszközhöz. Ahogy a Redux esetében láthattuk, az adatfolyam gráf felépítéséhez a legfontosabb a program által létrehozott adatok közötti kapcsolatok feltárása, azaz az adatok létrehozásához felhasznált korábbi adatok követése.

A Redux működése alapvetően a következő pontokban foglalható össze:

Memória és regiszterek „árnyékolása” A Redux a Valgrind által a memóriarekeszekhez és a regiszterekhez kapcsolt árnyékértékeket használja a tárolt adatok metainformációinak elérésére. Az árnyékértékek pointereket tárolnak, amelyek az adatokat leíró struktúrákra mutatnak.

Adatok reprezentálása, források rögzítése Az adatokat reprezentáló struktúrákban a Redux az adott adat kiszámításához felhasznált más adatokat leíró struktúrák referenciáit, az utasítás jellemzőit és egyéb információkat tárol.

Utasítások követése, gráf építése A program futása során a Redux az utasításokkal párhuzamosan, azoknak megfelelő módon építi fel az adatfolyam gráfot. Új adat

létrehozásakor új csomópontot hoz létre, meglévő adat másolásakor viszont csak a leíró struktúrára mutató referenciát másolja.

Az általam megvalósított adatfolyam kinyerési eljárás bizonyos szempontok szerint hasonló, más szempontokból azonban eltérő a Redux implementációjától. Az adatok reprezentálását a Redux-hoz hasonlóan végzem, az adatokat leíró objektumok tárolják a forrás adatok referenciáit és az utasítás jellemzőit. Az adatokat leíró objektumok tárolásához szintén a Redux-hoz hasonló árnyékolási technikát használtam, bizonyos eltérésekkel a memóriaterületek és a regiszterek reprezentációját tekintve. A felépített adatfolyam gráf szintén hasonló, abból a szempontból, hogy a gráfnak csak azon csomópontjai érhetőek el, amelyek a memóriában vagy a regiszterekben található adatokból bejárhatók. Érdekes megjegyezni, hogy amíg a Redux nem alkalmazott semmilyen szemétgyűjtési eljárást a már nem elérhető adatok által elfoglalt memória felszabadítására, addig az általam megvalósított rendszerben a Python szemétgyűjtő eljárása ezt automatikusan elvégzi.

A Redux bemutatásánál láthattuk, hogy a bájtos és félszavas műveletek kezelése milyen nehézségeket okozhat. Ott két megoldás merült fel, amelyek közül a Redux a bájtok kivágását és összefűzését használja. A feladat megoldása során egy ettől eltérő és mégis hasonló eljárást használtam. Ennek lényege – ahogy a későbbiekben részletesen bemutatom – hogy a rendszer a memóriarekeszek tartalmát bájtos felbontással követi, és minden esetben a végrehajtott műveletnek megfelelő bájtokat írja vagy olvassa. Ez a megoldás eredményét tekintve megfelel a Redux által használt kivágás és összefűzés alapú eljárásnak, azonban annál jóval egyszerűbben megvalósítható.

Az általam megvalósított rendszerben a felépített adatfolyam gráfon elvégzett feldolgozási lépések a Redux-nál alkalmazottaktól lényegesen különböznek. A Redux az adatfolyam felépítése után a kevésbé fontos csomópontok összevonásával és elhagyásával csökkenti a gráf méretét, hogy az megjeleníthető legyen. Az általam megvalósított alkalmazás ezzel szemben a gráf felépítése után az adatfolyamot teljes egészében adja át az aggregálást végző egységnek (l. következő fejezet).

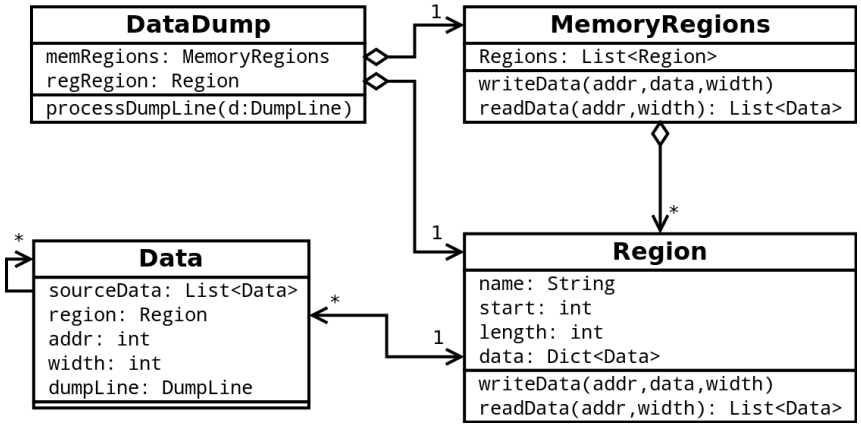
Amíg a Redux esetén a vizsgált program eredménye egyértelműen azonosítható a rendszerhívások által, a szimulált processzoros környezetben erre nincs lehetőség. Ennek megfelelően a program kimenetének tekintett memóriaterületet a vizsgálat elvégzése előtt kell megadni.

Az áttekintés után a tényleges megvalósítás részleteit az 5.9. ábra segítségével mutatom be. (Az ábrán csak a működés ismertetéséhez szükséges tagváltozókat és metódusokat tüntettem fel.)

DataDump

Az adatfolyam információk kivonása a `DataDump` osztályban történik. Az analízis program futása során a `DataDump` osztályból egyetlen példány jön létre.

Az osztály feladata a Giano napló alapján előálló `DumpLine` objektumok feldolgozása, és az adatfolyam gráf felépítése az azokban található információk felhasználásával. A rendszer a program futása közben előálló adatokat `Data` objektumokkal modellezi. A memóriában és



5.9. ábra. *Adatfolyam információk kinyerésével kapcsolatos osztályok*

a regiszterekben lévő adatokat a `Region` objektumok tárolják, amelyek részletes bemutatása a továbbiakban lesz olvasható.

A `DataDump` számára az utasításokat leíró `DumpLine` objektumokat egyesével kell átadni a `processDumpLine` metódus meghívásával. Mivel a feldolgozás menete függ az utasítás típusától, ezért elsőként ezt kell megvizsgálni. Az utasítások lényegében két csoportra oszthatók. A két csoportot a memóriareferens utasítások (load, store és ezek speciális esetei) és a memóriát nem használó utasítások képezik. A `DumpLine`-okban tárolt regiszter- és memória műveleteknek köszönhetően az adatfolyam felépítéséhez szükséges információk mindkét esetben könnyen meghatározhatók.

A memória műveleteket nem végző utasítások esetén a következő lépés az utasítás által felhasznált, azaz olvasott adatokat leíró `Data` objektumok összegyűjtése. Ehhez a `DataDump` a `DumpLine` `getReadRegOps` metódusát hívja meg, amely visszaadja a `regOps` tagváltozóban tárolt olvasási műveleteket. Miután így meghatároztuk, hogy mely regiszterek kerültek kiolvasásra, a forrás adatokat reprezentáló `Data` objektumok beolvashatók a `regRegion`-ból. Mivel regisztereken operáló műveletek minden esetben legfeljebb egy regiszterbe írják az eredményüket, a következő lépésben meg kell határozni ezt a cél regisztert. Ha a művelet generál kimenetet, azaz van írt regiszter, létrehozható az új adatot reprezentáló `Data` objektum, amelyben el kell tárolni többek között a forrás adatok referenciáit. Ezután az új adat beírható a cél regiszterbe a `regRegion` megfelelő metódusának meghívásával. Speciális esetet képeznek a konstans vagy immediate adatokat használó utasítások. Ezekhez az utasításokhoz a `DataDump` szintén létrehozza a `Data` objektumokat, azonban azok forrás listáját üresen hagyja, hiszen az új adat nem köthető egyetlen korábbi adathoz sem.

A memóriareferens utasítások esetén figyelembe kell venni a fentieken túl néhány további szempontot is a feldolgozás folyamán. A művelet irányától (load vagy store) és szélességétől (bájtos, félszavas, szavas) függően eltérő eljárás szükséges. Ezeken túl az egyszerre több adatot megmozgató load és store *multiple* utasítások is külön kezelendők.

A feldolgozási alapvető lépései ezekben az esetekben is megegyeznek a regiszterreferens utasításoknál leírtakkal, azaz az új adat forrásainak meghatározása, új `Data` objektum példányosítása és feltöltése, majd eltárolása. A regiszterek és a memória között több adat átmásolására használható load és store *multiple* utasítások feldolgozásakor például, mivel

az átmásolt adatok egymástól függetlenek, minden adathoz létre kell hozni egy `Data` objektumot. Érdemes megfigyelni, hogy amíg Redux az adatok puszta másolására szolgáló műveleteket nem rögzítette az adatfolyam gráfon, az általam készített alkalmazás ezekben az esetekben is létrehoz új adatokat. Ezt a megvalósítást az indokolja, hogy a párhuzamosíthatóság vizsgálata szempontjából ezek is fontos eseményeknek tekinthetők.

Region

A `Region` objektumok feladata a vizsgált program memóriaterületeinek és regisztereinek modellezése, vagyis a `Data` objektumok tárolása. A `Region` példányok tárolják az adott memória vagy regiszter terület nevét, kezdőcímét és hosszát, valamint az adott területre beírt `Data` objektumokat.

A `Region` osztály egységes megvalósítást biztosít a memória különböző részein (stack, heap, data) és a regiszterekben tárolt adatok modellezésére, ami önmagában is előnyös lehet az eszköz felépítésére vonatkozóan, de például a későbbiekben az eltérő architektúrához történő adaptálást is megkönnyítheti.

A `Data` példányokat a `Region` objektumok *Dictionary* tárolókban helyezik el, a gyors hozzáférés biztosítása érdekében kulcsként az adat címét felhasználva. A *Dictionary* által biztosított gyors beszúrás és keresés (nagyságrendileg O(1)) igen előnyös az analízis eszköz sebességét tekintve.

Az adatok írásához és olvasásához biztosított `writeData` és `readData` metódusok a cím és a beírandó adat mellet a hozzáférés szélességét (szavas, félszavas, bájtos) is paraméterként fogadják. A hozzáférés szélessége és címe alapján meghatározható, hogy az adott művelet mely bájtokhoz fér hozzá a memóriában. Írási műveletek esetén az átadott `Data` objektum mindegyik kijelölt bájt címre beíródik. Olvasáskor a `readData` hasonlóan a címzés és a szélesség által meghatározott bájtokat olvassa, azonban a kiolvasott `Data` objektumok közül csak az egyedieket adja vissza, azaz több azonos `Data` példány esetén az azonosak közül csak egyet-egyet. A metódus a `Data` példányokat minden esetben egy listában adja vissza.

Érdekes megfigyelni, hogy abban az esetben ha egy memóriaszóba több bájtos adatot írunk, a teljes szó kiolvasásakor keletkező adatnak – az egyetlen olvasás ellenére – több forrás-adata lesz. Ez elsőre ellentmondásosnak tűnhet, azonban ha belegondolunk, ebben az esetben valóban több különböző adat építi fel az eredményt.

Látható, hogy ezzel az egyszerű és hatékony megoldással a különböző adatszélességű memóriaműveletek jól kezelhetők.

MemoryRegions

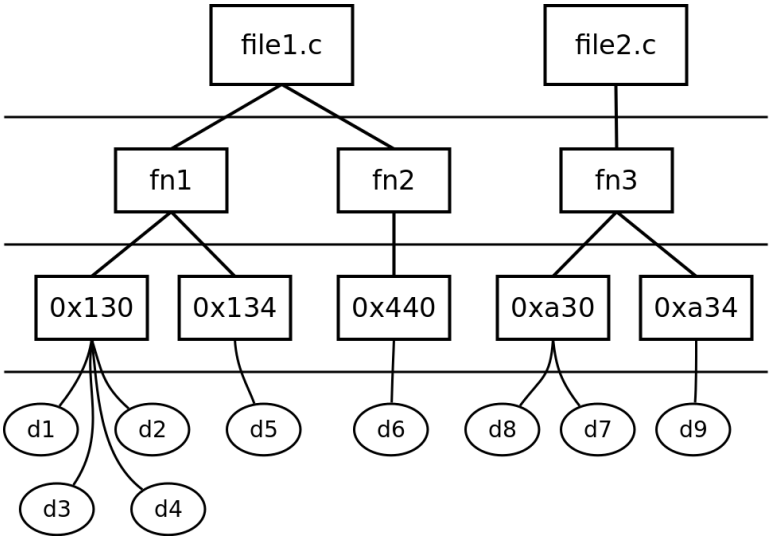
Az 5.9. ábrán látható `MemoryRegions` osztály, ahogy az ábrán is látható, több `Region` összefogására és együttes kezelésének lehetővé tételére szolgál. Ezt az osztályt az analízis eszköz a memória különböző régióinak (stack, heap, data) egységes kezeléséhez használja. A `MemoryRegions` legfontosabb feladata az írási és olvasási műveletek cím szerinti továbbítása a megfelelő `Region` felé.

Data

A `Data` osztály példányai az adatfolyam gráf csomópontjait, azaz magukat az adatokat írják le. A tagváltozók közül a legfontosabb a forrás adatokat reprezentáló `Data` objektumok referenciáit tároló `sourceData` lista. Az adat tárolási helyét (a regiszterekben vagy a memóriában) leíró `region` és `addr` tagok elsődleges szerepe a megjelenítés során lesz. A tárolt adat szélességét a `width` adja meg, amely szintén a megjelenítés szempontjából fontos. Az adatot létrehozó utasítást reprezentáló `DumpLine` objektum az utasítás fajtájának meghatározásához, illetve az időbeliség vizsgálatához szükséges.

5.4.3. Adatfolyam információk aggregálása

Az adatfolyam gráf aggregálásának legfőbb célja, hogy kinyerjük az összegyűjtött nagy mennyiségű információ *lényegét*. Ezt az aggregálás az adatfolyam gráf csomópontjainak és éleinek hierarchikus rendszerbe történő csoportosításával és összegzésével éri el. A folyamat során a hierarchikus kategorizálás alapját a vizsgált program részei (modulok, függvények, utasítások) adják. Az 5.10. ábrán ezek a hierarchia szintek figyelhetők meg. A legalsó szinten lévő adatokat először az azokat előállító utasítások (memóriacímmel szemléltetve), majd függvények, végül forrásfájlok szerint csoportosíthatjuk a felsőbb szinteken.



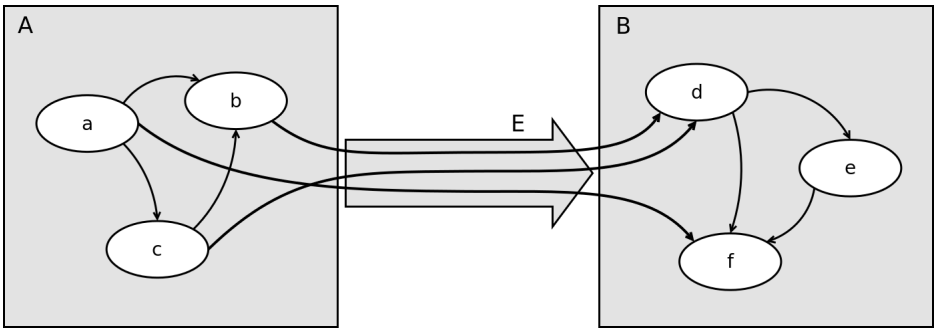
5.10. ábra. *Az aggregált adatfolyam gráf hierarchia szintjei*

A feldolgozás ennek megfelelően az előző lépésben előállított adatfolyam gráf csomópontjainak egy megadott halmazából indul, és az adatok közötti kapcsolatok mentén halad. A kiindulási halmazt praktikusan a program kimenetét alkotó memóriaterületen lévő adatok jelentik. Így az előálló aggregált adatfolyamon a kimenet előállításának folyamatát vizsgálhatjuk. A hierarchikusan végzett aggregálás előnye, hogy így a legalsó szinten található teljes adatfolyam gráfot különböző szinteken összegezve vizsgálhatjuk, a vizsgálat céljától függően.

Az aggregált adatfolyam reprezentálását több szinten egymásba ágyazott gráf csomópontokkal és élekkel valósítottam meg. A legalsó szinten található csomópontok közvet-

lenül a program futása során előálló adatoknak feleltethetők meg. A következő szinten a csomópontok a program utasításait reprezentálják, azaz ezek a csomópontok az azonos memóriacímen (programszámláló érték) található utasítások által létrehozott adatokat fogják össze. A további szinteken a csomópontok hasonló módon az azonos függvényekbe és azonos modulokba tartozó utasításokat illetve függvényeket egyesítik.

Az éleket, a csomópontokhoz hasonlóan, egymásba ágyazott struktúrák reprezentálják. Az 5.11. ábra az egy adott hierarchiaszinten megfigyelhető éleket szemlélteti. Az ábrán két



5.11. ábra. *Az aggregált adatfolyam gráf éleinek szemléltetése*

hierarchiaszint látható. A felsőbb szinten az `A` és `B` csomópontokat egyetlen `E` él köti össze. A felső szint csomópontjait jelölő négyzetekben az egy szinttel alacsonyabban lévő pontok és élek figyelhetők meg. Látható, hogy az `A` és `B` csomópontok és az `E` él is több elemre bomlik szét az alacsonyabb szinten.

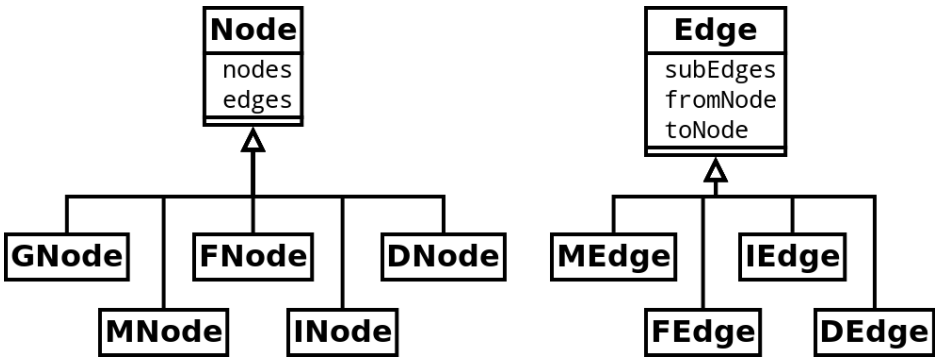
Az ábrán kétféle él figyelhető meg. Azok az élek amelyek egy felsőbb szintű csomópont két alpontját kötik össze, a felsőbb szintű ábrázolásban egyáltalán nem jelennek meg. A két felsőbb szintű csomópontban lévő két alpont között haladó élek ezzel szemben megjelennek a felső szint ábrázolásakor, ezek összesítése alkotja az `E` élet. A továbbiakban látni fogjuk, hogy ez a megkülönböztetés a gráf modellezése és a felépítése szempontjából is fontos. A gráf reprezentációját tekintve szintén lényeges megjegyezni, hogy a csomópontok önmagukban is gráfokként jelennek meg, hiszen további pontok és élek találhatóak bennük.

Node és Edge osztályok

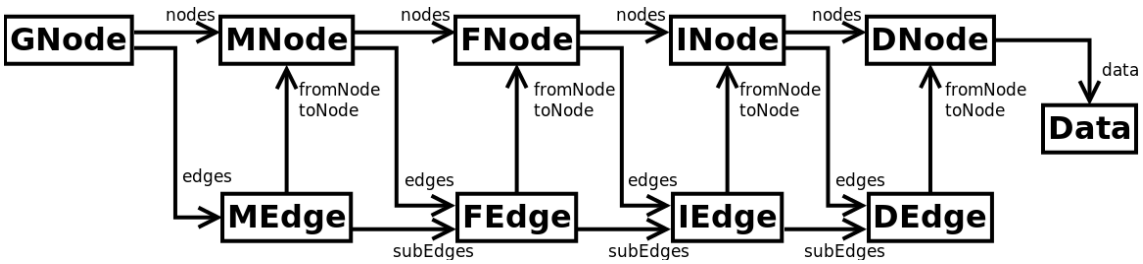
A következőkben a megvalósítás részleteit mutatom be. Az 5.12. ábrán az aggregált adatfolyam gráf csomópontjait és éleit modellező osztályok osztálydiagramja látható. Az osztályok közötti kapcsolatokat az 5.13. ábra mutatja.

A különböző szinteken található csomópontokat leíró osztályok mindegyike a `Node` osztály leszármazottja. Miután a csomópontok algráfokat tartalmaznak, a `Node` osztályban definiált `nodes` és `edges` ezeknek az algráfoknak a csomópontjait és éleit tárolják. Az élek esetén a közös attribútum az él két végpontján lévő csomópontok (`fromNode`, `toNode`) valamint az élekhez rendelhető al-élek referenciái (`subEdges`).

Az osztályok közötti kapcsolatokat bemutató ábrán megfigyelhető az egymásra épülő hierarchiaszintek reprezentációja. A legfelső szinten a `GNode` osztály áll, amely egyetlen csomópontként a teljes aggregált adatfolyam gráfot leírja. Az algráf elemeit ezen a szinten `MNode` és `MEdge` példányok alkotják. Ezek a példányok a program különböző moduljaiba



5.12. ábra. *Aggregált adatfolyam gráf csomópont és él osztályai I.*



5.13. ábra. *Aggregált adatfolyam gráf csomópont és él osztályai II.*

tartozó függvényeket (és azon belül utasításokat) egyesítik. Az ezen a szinten lévő élek ennek megfelelően a modulok közötti adatáramlásokat összesítik. Az `MNode`-ok az adott modulban, azaz forrásfájlban lévő függvényeket összesítő `FNode` csomópontokat foglalják magukba. A függvények közötti éleket az `FNode` példányok írják le.

A korábban említett kétféle típusú él reprezentációja közötti eltérés az eddig elmondottak alapján már megfigyelhető. Amíg az azonos modulban, azaz egy `MNode`-ban található két `FNode` példányt összekötő `FEdge` él az adott `MNode` él-listájában található, a két különböző modulban lévő két függvény közötti élet a modulokat összekötő `MEdge` al-élei (`subEdges`) között találhatjuk meg. Ez a megoldás bonyolultnak tűnhet, azonban a hierarchikus felépítés megragadása szempontjából ez az ideális, továbbá mind a gráf felépítése, mind pedig az ábrázolása egyszerű algoritmusokkal elvégezhető.

A további szinteken az azonos utasítások által létrehozott adatokat összefogó `INode`, valamint az egyes adatokat reprezentáló `DNode` osztályok találhatók. Az élek modellezése ezeken a szinteken is hasonló módon történik.

Az olvasóban felmerülhet, hogy az egymásba ágyazott csomópontokat és éleket leíró osztályok bemutatott megvalósítása túlságosan sok, egymáshoz nagyon hasonló alosztályt definiál, ami nem felel meg az objektumorientált szemléletnek. Az ilyen helyzetekben valóban alkalmazható lenne például a Kompozit (*Composite*) tervezési minta, amely fa-szerűen egymást tartalmazó objektumok modellezésére szolgál, azonban az aggregált adatfolyam gráfok leírásához ez nem lenne jól használható. Ennek oka, hogy itt az egyes hierarchiaszintek alapvetően rögzítettek, illetve a szinteken elvárt működés is különbözik. Az osztályok közötti jelentős hasonlóságokból adódó kód-ismétlések elkerülésére a *Template method* tervezési mintát használtam.

A *Template method* tervezési minta lényege, hogy a szülő osztályban definiált algoritmus bizonyos pontjain elvégzendő tevékenységeket a leszármazott osztályok határozzák meg. Így az algoritmus váza (az alosztályokban közös) a szülő osztályban található. A kritikus pontokon az algoritmusban metódushívásokat helyezünk el, amely metódusokat a szülő osztályban absztraktként definiálunk. A leszármazott osztályok ezek után az algoritmus működését ezeknek a metódusoknak a megvalósításával tudják befolyásolni. Ezzel a megoldással elkerülhető a felesleges kód-ismétlés, és a különböző alosztályok eltérő viselkedése is megvalósítható. A csomópontokat leíró `Node` osztályok esetén a *Template method* mintát többek között az új csomópontok létrehozásához használtam, mivel az új csomópont típusa az aktuális hierarchia szinttől függ.

A `Node` osztályok a bennük található al-csomópontok tárolására *Dictionary* tárolókat használnak, amelyek kulcs – érték párokat tároló asszociatív tömbök. A tárolás kulcsa minden hierarchia szinten az egyes csomópontokat megkülönböztető tulajdonság, vagyis az attribútum amely szerint az al-csomópontok összefogják a bennük található elemeket. Ennek megfelelően például az `MNode` példányok tárolása esetén a kulcs a modul neve, míg az `MNode` osztályban az `FNode`-ok tárolásának kulcsa a függvénynév. Az élek tárolásához szintén *Dictionary*-ket használtam, kulcsként az él által összekötött két node azonosítóját felhasználva.

A *Dictionary* tárolók használatának előnye a gyors beszúrás és keresés (megfelelően megválasztott kulcsok esetén), hiszen ahogy látni fogjuk a továbbiakban az aggregált adatfolyam gráf felépítésénél ezek a műveletek igen sokszor szükségesek.

A gráf felépítése

Ahogy már említettem, az aggregált adatfolyam gráf felépítése az adatfolyam gráf csomópontjainak (`Data` példányok) halmazából indul és az adatok forrás-adatai felé halad. A gráf felépítéséért a `DataDumpADFG` osztály felel. A példányosítás után a `beginDiscover` metódusnak adhatók át a kiindulási adatokat leíró `Data` objektumok. Ezután az adatfolyam gráf elemeinek bejárása mélységi keresés alapján történik.

A bejárás során elért minden `Data` objektumot el kell helyezni a hierarchikus kategorizálási rendszerben. Az elhelyezés a legfelső szintről indul, ahol az adatot előállító utasítást tartalmazó forrásfájlhoz tartozó `MNode` példányt kell kikeresni. Ha nem található az adott forrásfájlhoz `MNode` objektum, akkor azt létre kell hozni. Ezután a folyamat az `MNode` szintjén folytatódik. Itt az utasítást tartalmazó függvényt reprezentáló `FNode` példányt kell megkeresni vagy létrehozni ha nem létezik. A folyamat hasonlóan folytatódik egészen a `DNode` objektumok szintjéig, amelyek már az adatfolyam `Data` példányait tárolják.

Mivel az adott adathoz tartozó `Node` keresése illetve létrehozása alapvetően megegyezik a hierarchia szintek között, az ezt megvalósító `findOrCreateNode` metódust a `Node` osztály tartalmazza. A szintek közötti különbségek megragadásához itt is a *Template method* tervezési mintát vettem alapul.

Az aggregált adatfolyam éleinek létrehozása a csomópontok létrehozásával párhuzamosan történik, egy adott adat és forrás-adat párhoz tartozó `DNode` objektumok létrehozása

után a köztük lévő kapcsolatot reprezentáló él is létrehozásra kerül. Az élek esetén figyelembe kell venni a korábban bemutatott két él típus tulajdonságait. Az élek létrehozása szintén a legfelső szinten indul. Ha a két összekötendő adat (azaz `Data` példány) egy adott szinten azonos csoportba tartozik, azon a szinten nem kell élet létrehozni. Amikor a hierarchiában egy alacsonyabb szinten a két adat már különböző csoportokba esik (például két függvénybe), az ezt a két al-csomópontot összekötő élet létre kell hozni. A hierarchia további szintjein a két adatot tartalmazó csoportokat összekötő éleket ezután al-élekként kell létrehozni, az 5.11. ábrán látottaknak megfelelően.

A hierarchikus szerkezet felépítése során előfordulhat, hogy olyan `Data` példányhoz érkezünk el, amelyet már bejártunk. Ez például több kezdő adat megadása esetén lehetséges. A jelenség önmagában nem okoz problémát, azonban a feldolgozás idejét nagyban növelheti. Ennek elkerülése érdekében a `DataDumpADFG` nyilvántartja, hogy mely `Data` példányokat dolgozta már fel. Korábban már feldolgozott `Data` érzékelése esetén a `DataDumpADFG` az adott adat irányába nem folytatja tovább a mélységi keresést, hiszen az onnan elérhető részgráf `Data` objektumait már bejárta.

Nézetek előállítása

Az eddigiek alapján tehát előállt az adatfolyam gráf elemeinek hierarchikus kategorizálása. A szó legszorosabb értelmében ez az eredmény még nem tekinthető *aggregált* adatfolyam gráfnak, azonban annak létrehozásához már csak egy apró lépés szükséges, amely során egy kategorizálási szintet kiválasztva, és az alacsonyabb szinten lévő információkat összesítve egy ténylegesen aggregált nézetet készítünk.

A nézet létrehozásához az előzőekben létrehozott gráfból kigyűjtjük az egy adott hierarchia szinten lévő csomópontokat és éleket. Az élek esetén ismét figyelembe kell venni az al-éleket is úgy, hogy a legfelső szintről indulva a csomópontokban tárolt éleket és az élek al-éleit is bejárjuk.

Ennek az eljárásnak az eredménye az adott szint csomópontjainak és éleinek listája, amelyből a kimeneti DOT formátumú gráf-reprezentáció már előállítható. A kimeneti gráfon ekkor már feltüntethetőek az összesített értékek, amelyek csomópontok esetén például az adott csomópont alatt található `DNode` példányok, azaz létrehozott adatok számát mutathatja. Élek esetén hasonlóan a `DEdge` példányok számát jeleníthetjük meg. A csomópontok esetén az adatok száma hozzávetőleg arányos azzal, hogy az adott csoport (például függvény vagy modul) mennyi „munkát” végzett a kimenet előállításához. A több adatot tartalmazó csoportoknak nyilvánvalóan nagyobb szerepük van a végeredmény létrehozását tekintve.

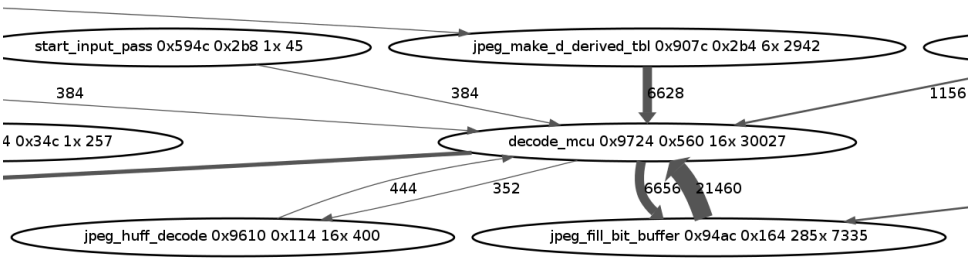
Az élek összesített mutatói szintén értékes információt hordoznak. Az összesített érték az adott élen áthaladó adatok mennyiségét mutatja, amely bizonyos értelemben a két összekötött csoport kapcsolatának szorosságát érzékelteti.

Ahogy a gráf megjelenítését végző Graphviz bemutatásánál (4.3. fejezet) kifejtettem, az előállított ábrázolások csomópontjainak elrendezése lényegében nem befolyásolható. Ez a korlát főleg nagy gráfok esetén jelent problémát, mivel ekkor általában az automatikus

elrendezés nem eredményez jól átlátható gráfot. A probléma enyhítésére az aggregált nézet előállítása során lehetőség van a kevésbé fontos élek és csomópontok elhagyására. A megadott határérték alatti összesített adatáramlással rendelkező élek eltávolítása után az így kapcsolatok nélkül maradt csomópontok is eltávolíthatóak. Az élek eltávolításának hatását a függelék F.1. fejezetében található ábrákon figyelhetjük meg részletesen.

A tapasztalataim alapján a megfelelő határértékkel elhagyott élek jellemzően mindössze néhány adat átvitelét mutatják, azonban a gráf megjelenítésekor igen zavaróak. Arra vonatkozóan, hogy az elhagyott élek által hordozott információ elvesztése hogyan befolyásolja a párhuzamosíthatóság elemzését, további vizsgálatok szükségesek.

A gráf ábrázolhatóságának javítására egy másik, valamivel hatékonyabb funkciót is megvalósítottam. Ennek lényege, hogy a gráf egy kiválasztott csomópontjának vizsgálatakor csak az adott csomópont és annak közvetlen szomszédai jelennek meg. A vizsgálandó csomópont a teljes gráf ábráján interaktívan, kattintással választható ki. Így a megjelenítendő gráf elemek száma jelentősen csökkenthető, és a vizsgálat a jobban átlátható ábrázolás miatt hatékonyabban elvégezhető. A következő fejezetben bemutatásra kerülő valós analízis során ezt a megjelenítési módot használtam.

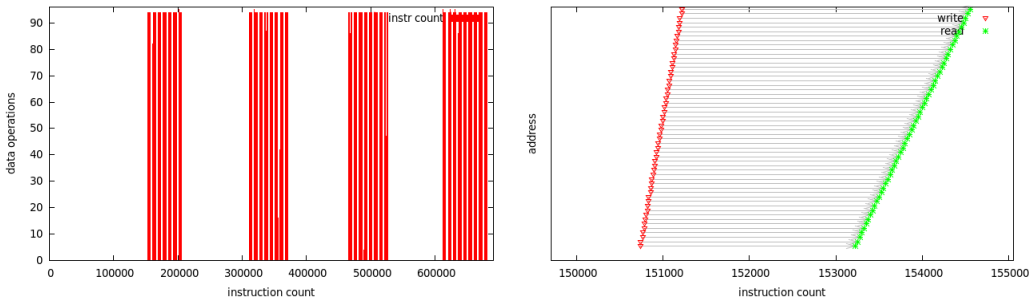


5.14. ábra. *Aggregált adatfolyam gráf részlet*

Az összesítéseket is mutató aggregált adatfolyam gráf részleten (5.14. ábra) egy vizsgált program függvény szintű nézete látható, azaz a csomópontok a program függvényeit, az élek pedig a köztük áramló adatokat jelenítik meg. Az élek vastagsága – a jobb áttekinthetőség érdekében – az átáramló adatmennyiséggel arányos. A függvényeket reprezentáló csomópontokban a függvények neve, memóriacíme és hossza, valamint a futások és a létrehozott adatok száma látható. Habár a függvények elnevezései esetében – a rövid megjeleníthetőség érdekében – nem látható azok hívási kontextusa, az ábrán a különböző kontextusokból hívott függvények külön csomópontként vannak ábrázolva. Ez a megoldás elsőre nem tűnhet logikusnak, azonban bizonyos esetekben célszerű az aggregált adatfolyam gráf elemeit kontextusuk szerint elkülönítve ábrázolni.

Időbeli és memóriacím szerinti nézetek

Az eddigiekben bemutatott összesített értékek az aggregált adatfolyam gráfban rögzített hatalmas mennyiségű információ csupán egy nagyon kis részét jelenítik meg. Az információ további részeinek megjelenítésével az analízis eszköz sokkal hatékonyabbá tehető.



(a) *Függvény futási hisztogramja* **(b)** *Adatáramlás egy kiválasztott élen*

5.15. ábra

A `Data` példányokban elérhető információk között megtalálható az adott adat létrehozási ideje és az általa elfoglalt memóriaterület címe. Ezeket az információkat felhasználva további két nézetet hozhatunk létre. A gráf csomópontjai esetén az adatok létrehozási időpontjainak hisztogram jellegű ábrázolásával az 5.15a. ábrán láthatóhoz hasonló ábrát kaphatunk. Erről az ábrázolásról leolvasható, hogy az adott függvény a program futása során milyen időpontokban volt aktív, mikor keletkeztek a benne található adatok.

Az élek esetén az él által összekötött két `Data` példányból kinyerhető az adott adat létrehozásának és felhasználásának ideje, valamint az adatot tároló memóriacím (vagy regiszter). Már önmagában a létrehozási és felhasználási idők alapján is igen sok következtetés levonható, azonban az időzítési és a memóriabeli elhelyezkedésre vonatkozó információkat kombinálva és egyesítve egy sokkal hatékonyabb ábrázolás hozható létre. Egy kiválasztott élre vonatkozó ilyen megjelenítés részlete az 5.15b. ábrán látható. A vízszintes tengely a program futásának idejét mutatja a végrehajtott utasítások szerint, a függőleges tengelyen pedig a memóriacím került felvételre. Az ábrázolt pontok mindegyike egy-egy adat létrehozását (*write*) vagy felhasználását (*read*) jelzi, annak ideje és címe szerint. A grafikon értelmezését vízszintes vonalak segítik, amelyek az írási és olvasási pontokat kötik össze.

Fontos kiemelni, hogy az ábrázolt adatok létrehozása és felhasználása két különböző függvényben történik, azaz ezzel a megjelenítéssel valóban a két függvény közötti adatáramlás vizsgálható nagyon nagy részletességgel.

A következő fejezetben további, valós példák láthatóak az ilyen megjelenítésekre, és azok értelmezésére.

6. fejezet

Az analízis eszköz működés közben

Ebben a fejezetben az elkészített analízis eszköz működését és használhatóságát egy valódi algoritmus elemzésével mutatom be. Ennek részeként az aggregált adatfolyam gráfok hatékonyságát is értékelem majd. Mindezekhez elsőként egy alkalmasan vizsgálható algoritmus kiválasztása szükséges, amely a tesztelés alapjául szolgál.

A fejezet első részében az algoritmus kiválasztásának szempontjait, a választott algoritmust és annak elméleti hátterét mutatom be. Ezután rátérek az aggregált adatfolyam gráf vizsgálatához szükséges előkészítő lépések ismertetésére, végül a vizsgált program gráfjának elemzésével értékelem az aggregált adatfolyam gráfok használhatóságát több szempont alapján is.

6.1. A vizsgált algoritmus

Az analízis eszköz kipróbálásához használt algoritmus kiválasztásánál a következő szempontokat vettem figyelembe:

Valós feladat A kiválasztott algoritmusnak egy valós, lehetőleg általánosan előforduló feladatot kell megoldania.

Forráskód szinten elérhető megvalósítás Mivel a vizsgált algoritmus megvalósítása semmilyen szempontból nem része a feladatnak, a kiválasztott algoritmushoz C nyelven írt megvalósításnak kell elérhetőnek lennie.

Könnyen skálázható futási idő Az analízis eszköz kipróbálásához nagyon előnyös ha a választott algoritmus futási ideje (végrehajtott utasítások száma) könnyen befolyásolható, hiszen többek között a feldolgozási idő és az utasítások száma közötti összefüggés nem ismert előre.

Csak fixpontos számítások Miután a szimulált ARM processzor nem rendelkezik lebegőpontos utasításokkal, az ilyen műveletek emulálása további problémákat vethet fel. Ezek elkerülése érdekében a választott algoritmus nem alkalmazhat lebegőpontos számításokat.

A fenti szempontok figyelembevételével az analízis eszköz valós helyzetben történő kipróbálásához egy JPEG kitömörítő algoritmust megvalósító programot választottam ki.

Ez a program forráskód szinten elérhető az *Independent JPEG Group*[26] által ingyenesen elérhetővé tett JPEG programkönyvtárban. A vizsgálatokhoz a programkönyvtár egy viszonylag régi, `6a` jelű változatát használtam. A programkönyvtár implementációja igen összetett, mivel az egyszerű dekódolás mellett sok egyéb funkciót is tartalmaz (például tömörítés, valamint sokféle ki- és bemeneti formátum). További előny, hogy a kód fixpontos számábrázolást alkalmaz, így nincs szükség lebegőpontos számításokra. Mivel az algoritmus működése legnagyobb részben a bemeneti képtől függ, a futási idő (az utasítások száma) nagyon jól skálázható a bemeneti kép méretének módosításával.

A következőkben elsőként röviden bemutatom a JPEG tömörítés működését, majd a JPEG adatok tárolásához használt fájlformátumot is.

6.1.1. A JPEG tömörítés ismertetése

A JPEG tömörítés részleteit a hivatkozott ISO/IEC szabvány rögzíti[27]. Ahogy a szabvány neve is tartalmazza (*„continuous-tone still images”*), a JPEG elsődlegesen a valóságban érzékelhető, folytonos jellegű változásokat tartalmazó képek tömörítésére alkalmas. A JPEG tömörítésnek van veszteségmentes változata is, de ez ritkábban használatos.

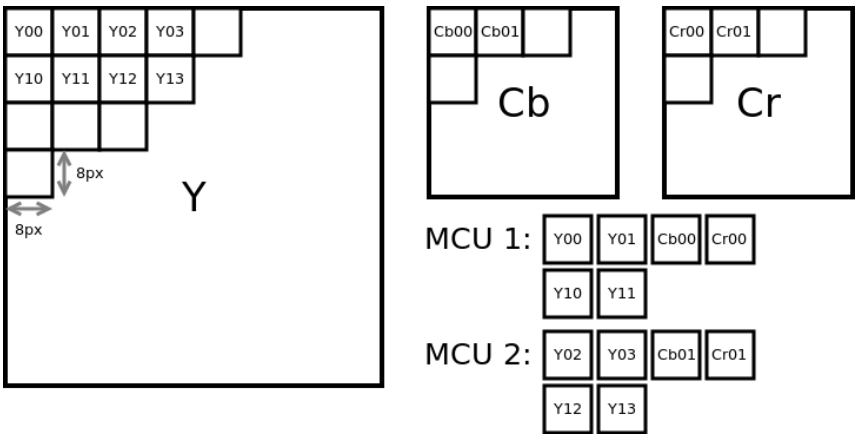
A JPEG által alkalmazott lépéseket elsőként a tömörítés esetén mutatom be nagyon röviden, a legegyszerűbb veszteséges esetben[27].

A tömörítés kiindulási adatait legtöbbször valamilyen egyéb, tömörítetlen képformátum tartalmazza. Az egyszerűség kedvéért feltételezzük hogy a bemenet pixelenként 3x8 bites RGB formátumban áll rendelkezésre.

A tömörítés első lépése a színtér transzformálása RGB reprezentációból Y'CbCr módba, amely elkülöníti a fényességi komponenst (luma) és két színkülönbségi jelet (chroma). A transzformáció lényegében egy mátrixszorzást, azaz lineáris kombinációt jelent. Az együtthatók a hivatkozott szabványban megtalálhatók[28]. Az átalakítás célja, hogy a fényességi és a színkomponenseket szétválasztva, majd azokat eltérő mértékben tömörítve, az eljárás hatékonyabb lehessen.

Kihasználva, hogy az emberi szem a színinformációkat lényegesen rosszabb felbontással érzékeli a fényességnél, a következő lépésben a két színkülönbségi csatorna felbontását lecsökkenthetjük. A tömörítés mértékétől függően lehetőség van a felbontás felére csökkentésére csak vízszintes, vagy vízszintes és függőleges irányban is. Természetesen az eredeti felbontás is használható a jobb minőség elérése érdekében.

A következő lépésben a tömörítendő képet blokkokra bontjuk, amelyek elnevezése MCU (*Minimum Coded Unit*). A színinformációk felbontásától függően egy MCU 8×8, 16×8 vagy 16×16 pixelnyi területet fedhet le az eredeti képből. A színinformációk csökkentése nélkül tehát az MCU-k a fényességi és a színcsatornák egy-egy 8×8 pixeles részét tartalmazzák. Vízszintes és függőleges, azaz négyszeres csökkentés esetén minden MCU a fényességi csatorna négy 8×8-as blokkját, míg a két csökkentett felbontású színcsatorna egy-egy 8×8-as blokkját tartalmazza. A 6.1. ábrán ezt az esetet szemléltettem. Megfigyelhető, hogy a két színcsatorna mérete csupán negyede a fényességi csatornának, és az MCU-k összeállítása is látható.



6.1. ábra. *JPEG* Minimum Coded Unit *blokkok*

A tömörítési eljárás következő lépése a kétdimenziós diszkrét koszinusz transzformáció (DCT) alkalmazása az MCU-k 8×8 pixeles blokkjaira[29]. A transzformáció lényege, hogy a pixelek információit térbeli frekvenciakomponensekre bontjuk. A transzformáció során a 64 bemeneti pixel érték alapján egy DC és 63 AC komponens keletkezik, amelyek a különböző térbeli frekvenciákat reprezentálják. A bemeneti adatokhoz hasonlóan a kimenetet is leggyakrabban egy 8×8-as mátrix írja le. A transzformáció egyik célja, hogy a frekvenciákat elkülönítve, lehetővé tegye azok eltérő mértékű tömörítését. Miután az emberi látás a nagyobb frekvenciájú változásokat kevésbé pontosan érzékeli, itt nagyobb mennyiségű információ hagyható el a minőség észrevehető romlása nélkül.

A tömörítés lényegében egyetlen veszteséges lépése a kvantálás, amely során a frekvenciakomponensek által hordozott információt csökkentjük komponensenként eltérő mértékben. A kvantálás mértékét megadó együtthatókat egy 8×8-as mátrix tartalmazza. Az összetartozó frekvenciakomponenseket és kvantálási együtthatókat egymással elosztva, majd az eredményt kerekítve kaphatjuk meg a kvantált értékeket. A kvantálási mátrix elemeire vonatkozóan a szabvány csak egy ajánlást tartalmaz, a ténylegesen alkalmazott értékek a megvalósítástól és a beállított tömörítési minőségtől is függhetnek. A legtöbb esetben igaz azonban, hogy a kvantálási tényezők a nagyobb frekvenciájú komponensek felé haladva egyre nagyobbak. A kvantálási mátrix elemei általában eltérőek a fényesség- és a színcsatornák esetén. A kvantálás eredményeként a frekvenciakomponensek egy része (jellemzően a nagy frekvenciákon) nullává válhat, ami a következő lépésben további tömörítést tesz majd lehetővé.

A tömörítés utolsó lépése az entrópia kódolás, amely során a kvantált frekvenciakomponens értékek tárolásához szükséges bitek száma jelentősen csökken. A kódolás alapja egy speciális lebegőpontos jellegű ábrázolási mód, amely a komponens tárolásához szükséges bitek számát (exponens) és a komponens értékét kódolja. A bitszámok tárolása esetén a további tömörítés érdekében Huffman kódolás is történik. A DC és AC komponensek tömörítése kis mértékben eltérő. Az AC együtthatók hatékonyabb tömörítése érdekében a tárolásukhoz szükséges bitek száma mellett az együtthatót megelőző nulla értékek száma is rögzítésre kerül. Ezzel még kevesebb biten tárolhatók az AC együtthatók között található

gyakori nulla értékek, amelyek csoportosulását az együtthatók cikk-cakk alakban történő kiolvasása is elősegíti. A Huffman kódoláshoz szükséges táblázatokra (a kódszavak és értékek összerendelése) vonatkozóan a szabvány tartalmaz ajánlásokat, azonban ezektől szintén el lehet térni.

A dekódolás során lényegében a kódolás lépései ismétlődnek fordított sorrendben. Elsőként vissza kell állítani az entrópia kódolt adatfolyamból a DC és AC együtthatókat. A kvantálás visszaállításához az együtthatókat egyesével össze kell szorozni a kvantálási mátrix megfelelő elemeivel, az eredeti értékek közelítéséhez. A kerekítés során elvesztett információ természetesen itt nem nyerhető vissza. A következő lépés a 8×8-as blokkokba rendezett együtthatók inverz koszinusz transzformációja, amely eredményeként már az eredeti pixel-információk közelítését kapjuk. Az egyes MCU egységekbe tartozó blokkok összerendezése után (l. 6.1. ábra) szükség esetén elvégezhető a színinformációk felbontásának visszaállítása (valamilyen interpolálási eljárással). A dekódolás utolsó lépéseként, a fényességi és színkülönbségi csatornák RGB színtérbe transzformálásával, előáll a kép megjeleníthető vagy valamilyen fájlformátumban eltárolható változata.

JPEG fájlformátum

A JPEG tömörített képek tárolására a szabvány az *JPEG Interchange Format (JIF)* formátumot definiálja. Ebben a formátumban a tényleges képi információk mellett megtalálhatóak a tömörítés során használt kvantálási mátrixok és Huffman kód táblázatok, amelyekre a dekódolás elvégzéséhez szükség van.

A JIF formátum egyes részeit markerek különítik el. Ilyen markerek jelzik a fájl és a kódolás pontos típusát, a Huffman kódtáblák és kvantálási mátrixok definícióit, valamint a tényleges tömörített adatfolyamot. A szabványban nem rögzített, kiegészítő információk tárolására is van lehetőség speciális markerek használatával.

6.2. A vizsgálat előkészítése

A JPEG kódolás rövid elméleti áttekintése után ebben az alfejezetben bemutatom, hogy milyen előkészítő lépések elvégzésére volt szükség a kitömörítő algoritmus vizsgálata előtt.

6.2.1. Az operációs rendszer hiányából adódó problémák megoldása

Mivel a vizsgálat alapja a Giano által szimulált ARM 7-es processzor, a vizsgált algoritmus lefordításához kereszt-fordítót kell használnunk, ahogyan ezt korábban már említettem. A kereszt-fordító önmagában csak azt biztosítja, hogy a fordítás eredményeként keletkező utasításokat a processzor értelmezni fogja, a program működéséhez azonban ez nem elegendő. A probléma eredete, hogy míg a vizsgált JPEG megvalósítás alapvetően PC-n, operációs rendszer alól történő futtatáshoz készült, a vizsgálati környezetben közvetlenül a processzoron kell futnia.

Az operációs rendszer támogatás nélkül való futtatás által felvetett problémákat és azok megoldását a következőkben foglalom össze:

Dinamikus memóriakezelés

A vizsgált JPEG megvalósítás által használt dinamikus memória foglalás operációs rendszer nélkül nem működőképes. A probléma megoldását némileg leegyszerűsítette, hogy a JPEG implementáció a memória allokáláshoz egy belső függvényt definiál, amely továbbítja a kéréseket a `malloc` felé. Ezt a függvényt úgy módosítottam, hogy a `malloc` függvény meghívása helyett egy általam elkészített minimális memória allokáló algoritmust hajtson végre. Ez az algoritmus egy kellően nagy, statikus tömbből osztja ki a szükséges méretű területeket folyamatosan, a memória felszabadítások figyelmen kívül hagyásával. Fejlettebb memóriakezelés is megvalósítható lenne, azonban mivel a szimulált memória nem szűkös erőforrás, valamint a vizsgált program az allokált memóriaterületeket csak a futás végén szabadítja fel, ez a megoldás kielégítő.

Fájl I/O

A JPEG implementáció a be- és kimeneti adatok (képek) kezeléséhez fájlműveleteket használ, amelyekhez szintén az operációs rendszer támogatása szükséges. A probléma egyszerű, gyors és nem túl elegáns megoldásaként megkerestem a forráskódban a megfelelő be- és kimeneti fájlok olvasását és írását végző `fread` és `fwrite` függvényhívásokat, majd kicseréltem azokat saját függvényekre mutató hívásokra. Ezekben a függvényekben – miután a szimulált rendszeren csak a RAM áll rendelkezésre – a bemeneti fájlt egy statikusan feltöltött, globális tömbből olvasom, míg a kimenetet szintén egy globális tömbbe írom. A fájlokat megnyitó és lezáró függvényhívásokat ezután egyszerűen eltávolítottam.

Parancssori argumentumok

A parancssori argumentumok esetén szintén egy igen egyszerű megoldást alkalmaztam. Mivel a megfelelő működéshez néhány paramétert át kell adni, ezeket a `main` függvény elején sztringek formájában definiálom, majd az ezekre mutató pointerekkel töltöm fel az `argv` tömböt. Ezzel a minimális módosítással a program a megszokott módon dolgozhatja fel az argumentumokat.

glibc OS hívások

A standard C könyvtári függvényeket megvalósító `glibc` szintén erősen támaszkodik az operációs rendszerre, néhány alacsony szintű függvényhíváson keresztül. Hogy a fordítás során ezek hiánya ne okozzon gondot, ezen függvények minimális megvalósítását is biztosítanom kellett. Ilyen függvény például a dinamikus memóriakezeléssel kapcsolatos `sbrk`, amelyet elsődlegesen a `malloc` használ. Mivel a vizsgált programból eltávolítottam a `malloc` hívásokat, elegendő egy üres `sbrk` függvényt biztosítani.

Ellenőrzésképpen a fentieknek megfelelően módosított forráskódot ezután a keresztfordító mellett lefordítottam a „normál” GCC használatával is. Így megkaptam a program PC-n futtatható változatát, amelyet ezután lefuttattam. A futtatás végén a kimeneti képet tartalmazó memóriaterületet debugger segítségével egy fájlba írtam. Miután megbizonyosodtam arról, hogy az így keletkezett kimenet megegyezik az eredeti program által generált kimenettel, rátérhettem a szimuláció elvégzésére és az eredmények feldolgozására.

6.2.2. A fordítás, linkelés és futtatás lépései

A megoldás során a Mentor Sourcery CodeBench megfelelő változatát használtam. A fordításhoz így az `arm-none-eabi-gcc` parancsot kell meghívni. A parancssori argumentumok közül kiemelném a cél processzor típusát megadó `-mcpu=arm7`, és a saját indítási kód (*startup files*) megadásának lehetőségét biztosító `-nostartfiles` kapcsolókat.

Mivel a szimulált rendszer csak RAM memóriával rendelkezik, a rendszer indítása igen egyszerűen elvégezhető. (Ellentétben például a mikrokontrollereknél általános felépítéssel, ahol az indításkor a globális változók kezdőértékeit át kell másolni a FLASH memóriából a RAM területre.) A szimuláció kezdetén a processzor a beállított nullás kezdőcímről kezdi az utasítások végrehajtását. Ezen a címen a processzor reset vektora található. A vektor tábla tartalmának beállításáért a `startup.s` assembly fájl felelős. A nullás címen található ugró utasítás a program futását a `Reset_Handler` cimkéhez továbbítja, ahol megtörténik a kezdő stack pointer beállítása, majd a vezérlés átadódik – az immár C-ben definiált – `main` függvénynek.

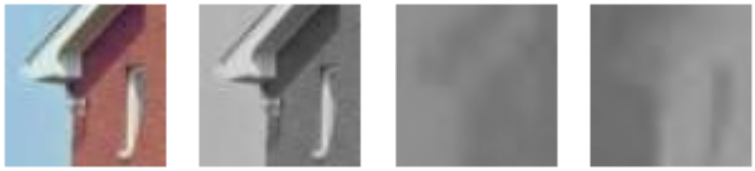
A program linkelését meghatározó linker szkript biztosítja, hogy a fordítás során a vektor táblát alkotó utasítások a megfelelő memóriacímre kerüljenek. A linker szkriptben ezen túl az egyes memória régiók (stack, heap, data) elkülönítéséhez szükséges szimbólumok definíciói is megtalálhatók, amelyek az analízis eszköz számára teszik lehetővé a régiók határainak automatikus azonosítását.

A fordítási és linkelési lépések után keletkező bináris fájl ezután betölthető a Giano szimulátorba, majd a futást leíró naplófájlok továbbíthatóak az analízis eszköz felé.

6.3. Algoritmus elemzése aggregált adatfolyam gráfok használatával

A következőkben a vizsgált JPEG dekódoló algoritmus felépítésének és működésének áttekintő bemutatása olvasható. Az elemzés közben az aktuálisan bemutatott részekhez tartozó aggregált adatfolyam gráfokat is megvizsgálom.

A vizsgálathoz egy 64×64 pixeles, JPEG tömörített képet használtam. Ez a meglehetősen kis méretű kép az analízis eszköz futási idejének és memóriahasználatának elfogadható szinten tartása miatt volt szükséges. A 6.2. ábrán megfigyelhető a vizsgált kép, amelynek eredeti forrása az *Independent JPEG Group* (IJG) programkönyvtára. Az eredeti képből a GIMP képszerkesztő programmal vágtam ki a vizsgált részletet. A beállított minőségi



6.2. ábra. *A vizsgálathoz használt JPEG kép és annak fényesség (Y) és színkülönbségi (Cb, Cr) csatornái*

szint mellett – ahogyan a későbbiekben látni is fogjuk – a GIMP a színcsatornák esetén függőleges és vízszintes irányban is felezi a felbontást.

A továbbiakban, a program működésének bemutatása során a legfőbb célom az aggregált adatfolyam gráf használatának és hatékonyságának szemléltetése. Ennek megfelelően a JPEG megvalósítását csak olyan mélységig ismertetem, ami az aggregált adatfolyam gráf elemeinek megfelelő azonosításához és megértéséhez elegendő.

A vizsgálat során bizonyos esetekben az analízis eszköz által előállított Callgrind kimenetet is felhasználtam, amelyre ezekben az esetekben külön ki fogok térni.

Fontos megjegyezni, hogy a továbbiakban vizsgált aggregált adatfolyam gráf minden esetben a program kimeneti adataira, azaz a dekódolt képre került felvételre.

6.3.1. Inicializálás

Ebben az alfejezetben a JPEG megvalósítás inicializálási lépéseit követem végig. Miután a teljes programkönyvtár igen széles funkcionalitású, a felépítése ennek megfelelően komplex. A forrásfájlok és almodulok sokasága sem könnyíti meg a részegységek közötti kapcsolatok és a működés megértését.

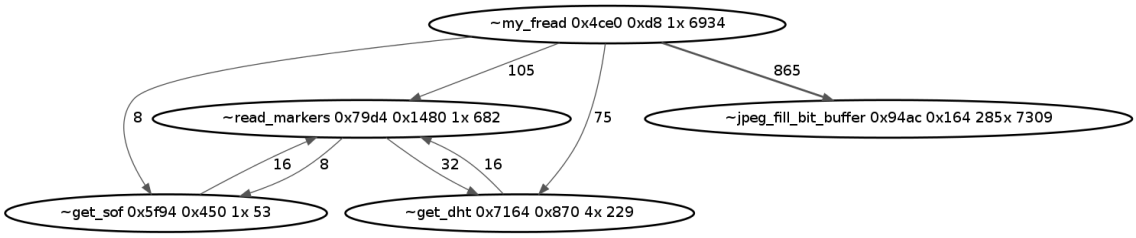
A kitömörítési folyamat középpontjában a `jpeg_decompress_struct` struktúra áll, amely a `jpeglib.h` fájlban definiált. Ebben a struktúrában megtalálhatóak többek között a kitömörítés alatt lévő kép információi (például méret, színtér), a kitömörítés folyamatát befolyásoló beállítások és paraméterek, a kvantálási mátrixok és Huffman táblák, valamint a kitömörítés során használt almodulok adatait tartalmazó struktúrákra mutató pointerek. Ilyen almodulok felelnek például a JPEG markerek feldolgozásáért, az inverz DCT transzformáció elvégzéséért vagy a színkülönbségi csatornák interpolálásáért.

A JPEG megvalósítás áttekintését a `djpeg.c` fájlban található `main` függvénnyel kezdem. A `jpeg_decompress_struct` típusú `cinfo` struktúra alaphelyzetbe állítását a `main` által meghívott `jpeg_create_decomperess` függvény hajtja végre. Itt történik a bemenetet és a JPEG markereket feldolgozó modulok inicializálása is.

A következő lépés a parancssori argumentumok értelmezése és a `cinfo` struktúra elemeinek megfelelő beállítása. Ezt a feladatot a `parse_switches` függvény végzi el. A parancssori kapcsolók használatával beállíthatóak többek között a ki- és bemeneti fájlok, a kimeneti formátum valamint a használni kívánt DCT megvalósítás.

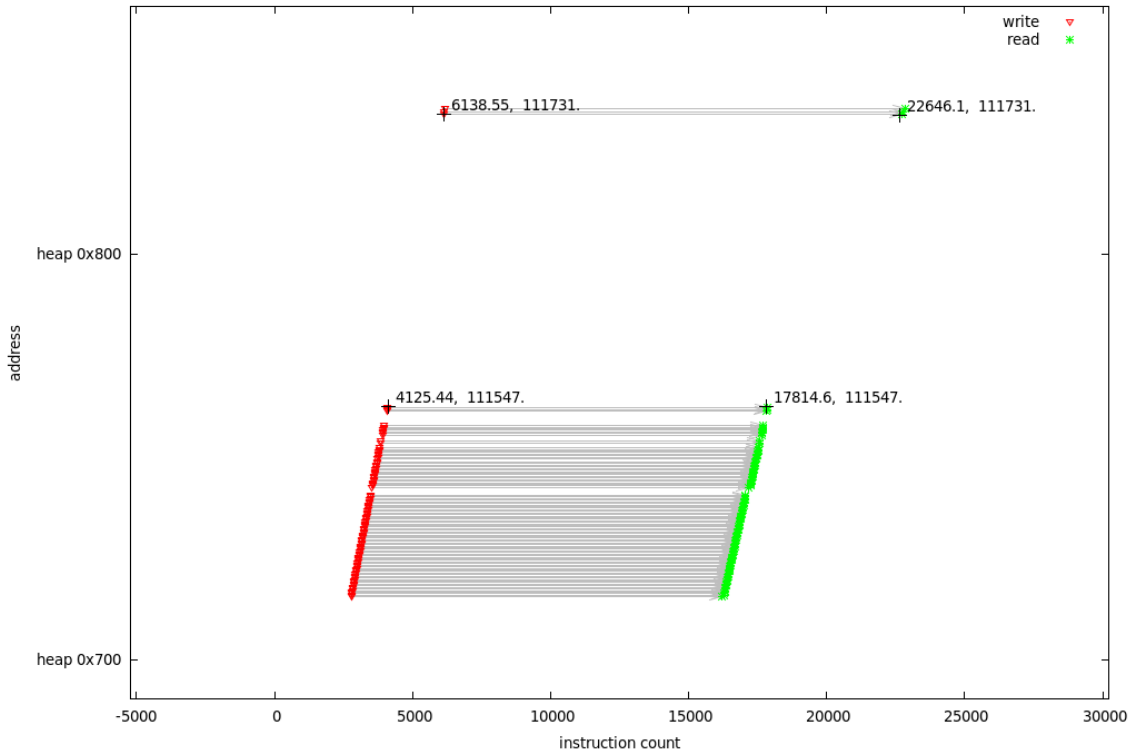
Az argumentumok feldolgozása után a bemeneti fájl megnyitása következik. (Ez a lépés a vizsgálat során természetesen nem hajtódik végre, a korábbiakban már bemutatottaknak megfelelően.) A bemeneti fájl beolvasásának előkészítése a `jdatasrc.c` fájlban található `jpeg_stdio_src` függvényben történik. Itt a legfontosabb lépés a bemeneti adatokat tároló 4096 bájtos buffer allokálása, és a beolvasást végző függvényre mutató pointer beírása a `cinfo` struktúrába.

A `main` függvényben meghívott `jpeg_read_header` függvény ekkor beolvassa és feldolgozza a JPEG fájl fejlécének tartalmát. Miután itt van szükség először a fájl tartalmára, ebben a függvényben történik meg az előzőekben látott buffer feltöltése, amely a `fill_input_buffer` függvény feladata. Ezután a `read_markers` függvény a buffer olvasásával megkeresi a fejléc részeit jelző markereket. A különböző részek tartalmának értelmezését és eltárolását a `read_markers` által meghívott függvények végzik.



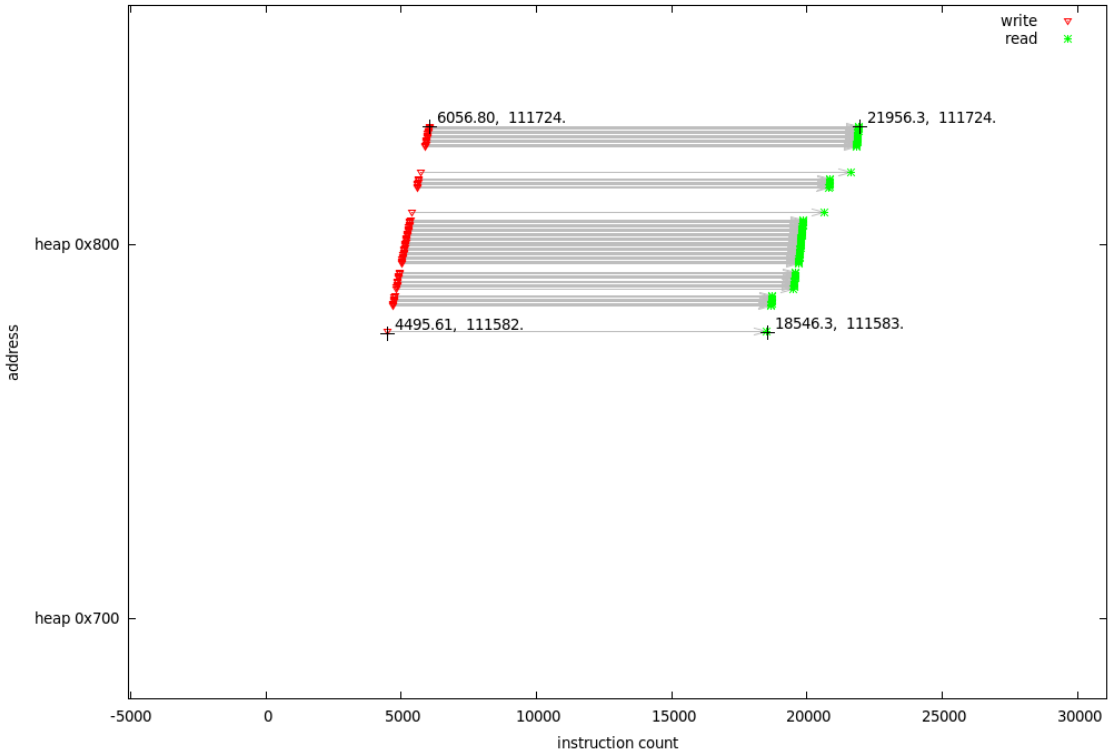
6.3. ábra. *A JPEG fájl adatainak beolvasása és felhasználásai*

Az aggregált adatfolyam gráf idevágó részletét a 6.3. ábra mutatja. Az aggregált adatfolyam jól mutatja, hogy a `my_fread` által a bufferbe beolvasott adatokat mely másik függvények használják fel. Megfigyelhető például, hogy a markereket feldolgozó `read_markers` függvény és az általa meghívott `get_dht` függvény is felhasznál adatokat. Az elvárásoknak megfelelően az adatok legnagyobb része a `jpeg_fill_bit_buffer` függvényben kerül felhasználásra, amelynek bemutatására még visszatérek.



6.4. ábra. *A* my_fread *és a* read_markers *függvények közötti adatáramlás*

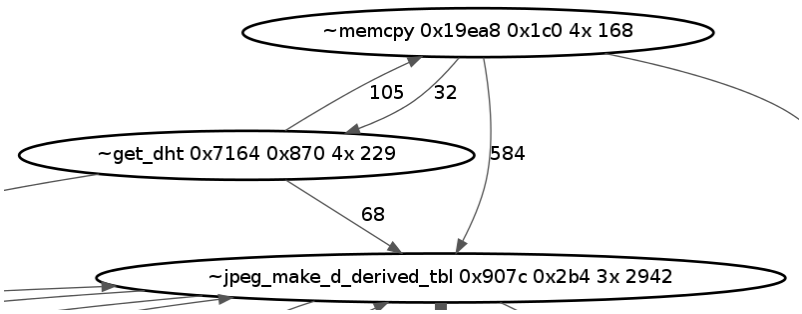
Az adatáramlást időzítés és memóriacím szerinti jellemzőit a 6.4. és a 6.5. ábrák mutatják. (Az ábrákon a vízszintes tengelyen a program utasításainak végrehajtása (idő), míg a függőleges tengelyen a memóriacím látható. Az egyes írási vagy olvasási műveleteket pontok mutatják, amelyek azt jelzik, hogy az adott időpontban az adott memóriacímet a program írta vagy olvasta.) Megfigyelhető, hogy a bufferbe beírt adatokat először a `read_markers` függvény olvassa, majd a DHT marker hatására vezérlés átkerül a Huffman táblákat beolvasó `get_dht` függvényre, amely folytatja az adatok olvasását. A `get_dht` visszatérése után láthatóan ismét a `read_markers` olvas néhány bájtot.



6.5. ábra. *A* my_fread *és a* get_dht *függvények közötti adatáramlás*

A főprogramba visszatérve, az inicializáció következő lépése a kimeneti modul példányosítása. A vizsgált esetben a kimeneti formátum a PPM (*Portable Pixmap Format*) volt. Ez egy egyszerű tömörítetlen formátum, amely színes (RGB) képek tárolásához használható. A `wrppm.c` fájlban lévő `jinit_write_ppm` függvény inicializálja a szükséges struktúrát, majd allokálja az egy sornyi pixel adat (azaz 3×64 bájt) tárolásához szükséges buffert. A kimeneti fájl fejlécének összeállítása és kiírása is ekkor történik meg.

A tényleges kitömörítés előtti utolsó lépés a szükséges almodulok inicializálása, amely általában a modulokhoz tartozó struktúrák allokálásából és azok kitöltéséből áll, bizonyos esetekben azonban további lépések is szükségesek. A dekódolási folyamat végén található színtér transzformáció gyorsabb végrehajtását elősegítő táblázat is itt kerül felépítésre. Ez a táblázat – ahogy a későbbiekben bemutatom – a transzformáció során végrehajtandó szorzási műveletek eredményeit tárolja az összes lehetséges bemenet esetére (4×256 bájt).



6.6. ábra. *A Huffman kódszavak dekódolását gyorsító táblázatok előállítása*

A Huffman kódszavak értelmezésének gyorsításához használt táblázatok is itt állnak elő. A `jpeg_make_d_derived_tbl` függvény, ahogy a 6.6. ábrán látható, a Huffman táblákat tartalmazó DHT szekciót feldolgozó `get_dht` függvénytől fogad adatokat, közvetlenül és egy memóriamásolási műveleten keresztül. Az előállított táblázatok felhasználását a későbbiekben láthatjuk majd.

Miután az összes szükséges egységet a megfelelő állapotba hoztuk, a következő lépés a tényleges pixel adatok dekódolása.

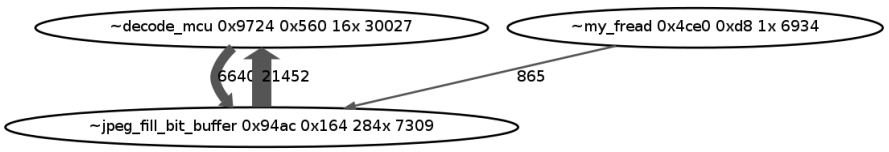
6.3.2. Dekódolás

A tényleges dekódolás szintén igen komplex megvalósítású. A `main` függvény által egy ciklusban meghívott `jpeg_read_scanlines` függvény már a teljesen dekódolt kép egyes soraival tér vissza. A paraméterként átadott sorbuffer pointer közvetlen kapcsolatot biztosít a PPM fájlt író modul felé.

A JPEG kitömörítés során előforduló nagy számú pointereken keresztüli függvényhívás követését nagyban megkönnyítette az analízis eszköz által előállított Callgrind kimenet, hiszen a dinamikus hívási gráfon a ténylegesen meghívott függvények láthatóak.

A `jpeg_read_scanlines` némi hibakezelés után továbbítja a paramétereket a `jdmainct.c` fájlban található `process_data_context_main` függvénynek. Ez a függvény – illetve modul – felelős a dekódolást végző függvények számára szükséges kontextus (például interpolálás esetén a szomszédos sorok) biztosításáért. A működés lényege, hogy egy teljes MCU sor (ami jelen esetben 16 pixel magas) dekódolása után a színkomponensek interpolálása és a színtér transzformáció már soronként történik meg. Az MCU-k dekódolását és bufferbe írását a `decompress_onepass` (`jdcoefct.c`) végzi. Ennek első lépése az entrópia kódolt adatfolyam visszafejtése, amit a `jdhuff.c` fájlban található `decode_mcu` függvény végez el.

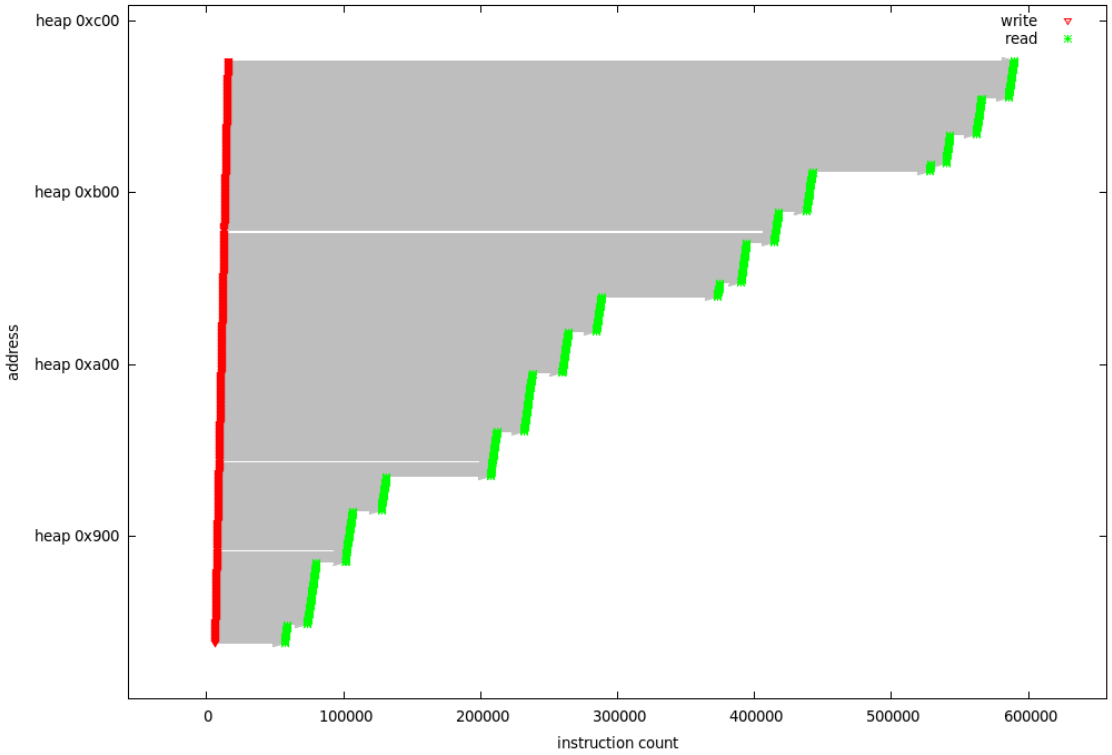
A hatékonyabb működés érdekében a `decode_mcu` elvárja, hogy a dekódolás eredményeként előálló DCT együtthatókat tároló memóriaterület nulla adatokkal legyen feltöltve. Így csak a nullától eltérő együtthatókat kell kiírni. Ezt a `decompress_onepass` függvény által meghívott `jzero_far` végzi.



6.7. ábra. *Bit buffer feltöltése a Huffman kódolás visszafejtéséhez*

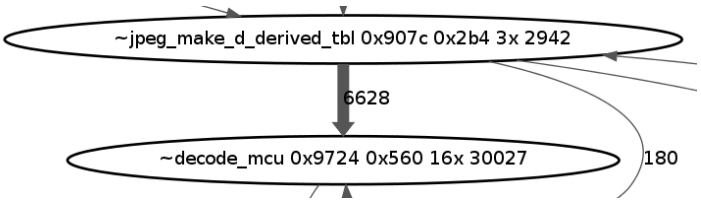
Az aktuálisan dekódolt biteket egy 32 bites buffer tárolja, amelynek adatokkal való feltöltéséhez a `decode_mcu` a `jpeg_fill_bit_buffer` függvényt hívja. Amint a 6.7. ábrán látható adatfolyam részleten megfigyelhető, az adatok közvetlenül a `my_fread` által írt globális bufferből származnak. A `decode_mcu` és a `jpeg_fill_bit_buffer` között megfigyelhető adatáramlás nagy része nem valós, azt csak a gyakori függvényhívások által

elmentett majd visszaállított regiszterek okozzák. A 6.8. ábrán a `my_fread` által feltöltött buffer olvasásának időbeli lefolyása látható. Jól megfigyelhető az MCU sorokhoz és egyes blokkokhoz tartozó adatok elhelyezkedése. (Egy MCU blokk egy egybefüggő olvasási szakasz, amit kis szünet követ.)

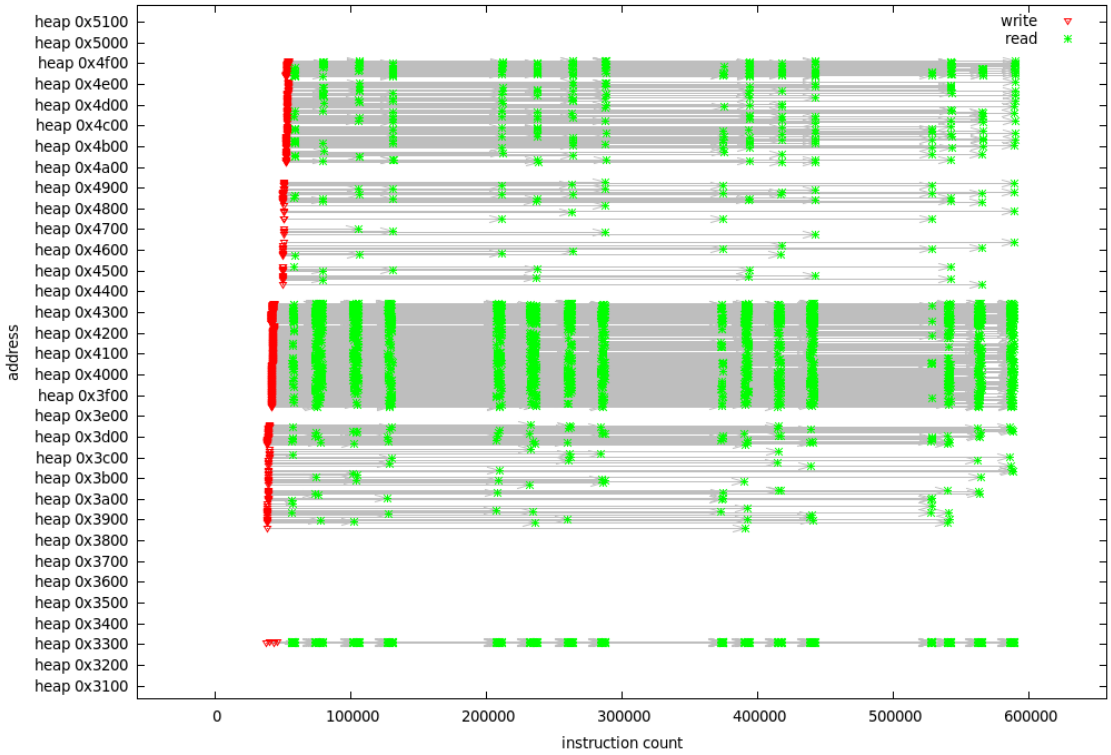


6.8. ábra. *Adatáramlás* my_fread *és* jpeg_fill_bit_buffer *között*

A Huffman kódolt adatfolyam feldolgozásához a korábban a `jpeg_make_d_derived_tbl` által előállított táblázatok is felhasználásra kerülnek, ahogy azt a 6.9. ábra is mutatja. Az itt tapasztalható – igen véletlen jellegű – adatáramlás memóriacím és idő szerinti ábrázolása a 6.10. ábrán figyelhető meg. A 6.8. ábrán látottakhoz hasonlóan itt is észrevehető a kép blokkjainak elkülönülő feldolgozása.

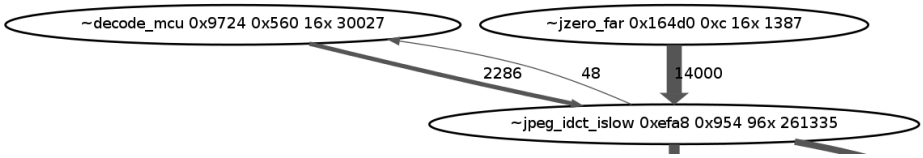


6.9. ábra. *A Huffman kódszavak dekódolását gyorsító táblázatok felhasználása*



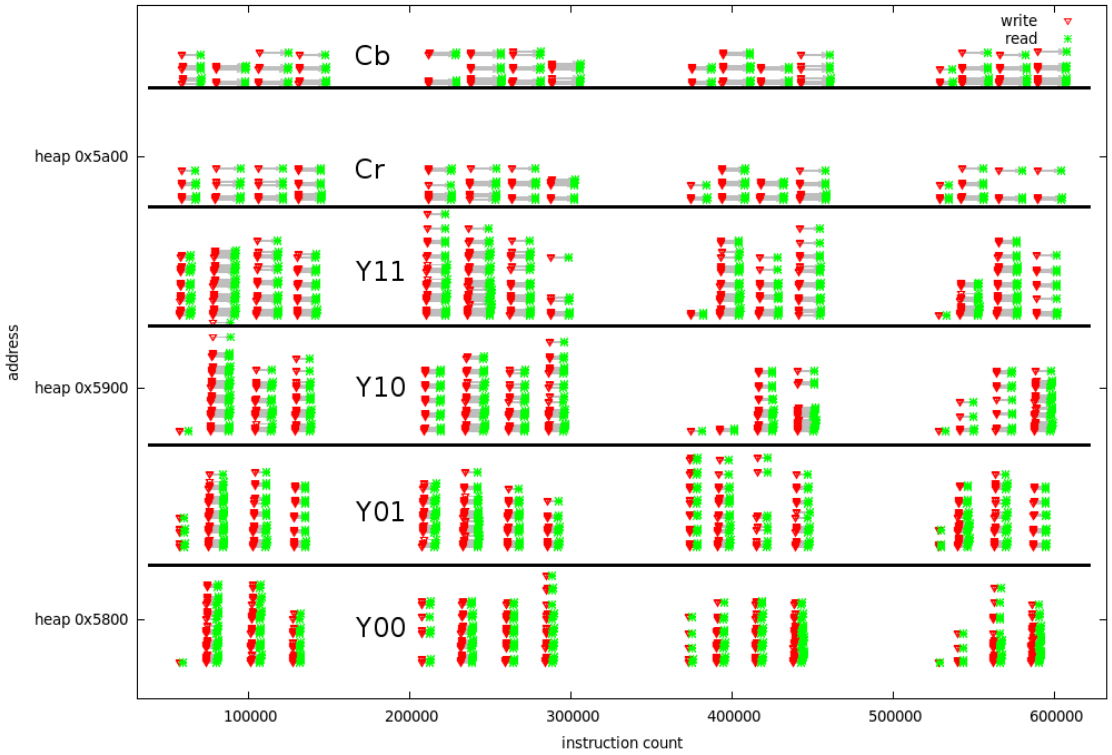
6.10. ábra. *A Huffman kódszavak dekódolását gyorsító táblázatok olvasásának időbelisége*

Az egyes MCU-k visszafejtése után következhet az azokban lévő 8×8 pixeles blokkok inverz koszinusz transzformációja. Az ezt a feladatot végző `jpeg_idct_islow` függvény által felhasznált adatok forrásait a 6.11. ábra mutatja. Amint az látható, a transzformálandó



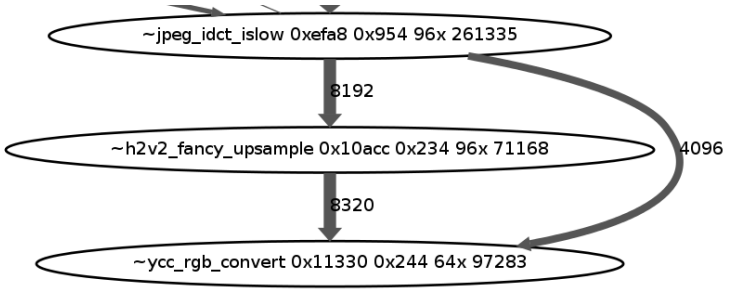
6.11. ábra. *Inverz koszinusz transzformáció bemenetei*

együtthatók egy része a `decode_mcu` függvény felől érkezik, a nagyobb része azonban – mivel a nulla együtthatókat a `decode_mcu` nem írja – a `jzero_far` függvényben jön létre. A 6.12. ábrán a `decode_mcu` által előállított együtthatók felhasználásának időbeli és memóriacím szerinti megjelenítése látható. Az idő szerint tapasztalható 4×4-es csoportosulás itt is az MCU-k elkülönülése miatt tapasztalható. Érdekes megfigyelni, hogy az írási és olvasási műveletek memóriacím szerint hat sorra oszthatók. Ennek oka, hogy minden MCU hat DCT blokkot tartalmaz, amelyek közül négy – ahogy az ábrán is jelöltem – a fényesség információkat, míg további kettő a színkülönbségi adatokat tartalmazza. Látható továbbá, hogy a két színkülönbségi csatorna esetén jóval kevesebb adat, azaz DCT együttható áll rendelkezésre (valószínűleg az erőteljesebb kvantálás következtében), valamint az egyes blokkok között is jelentős eltérések láthatók. Fontos kiemelni, hogy a 6.12. ábrán csak a nem nulla együtthatók jelennek meg.



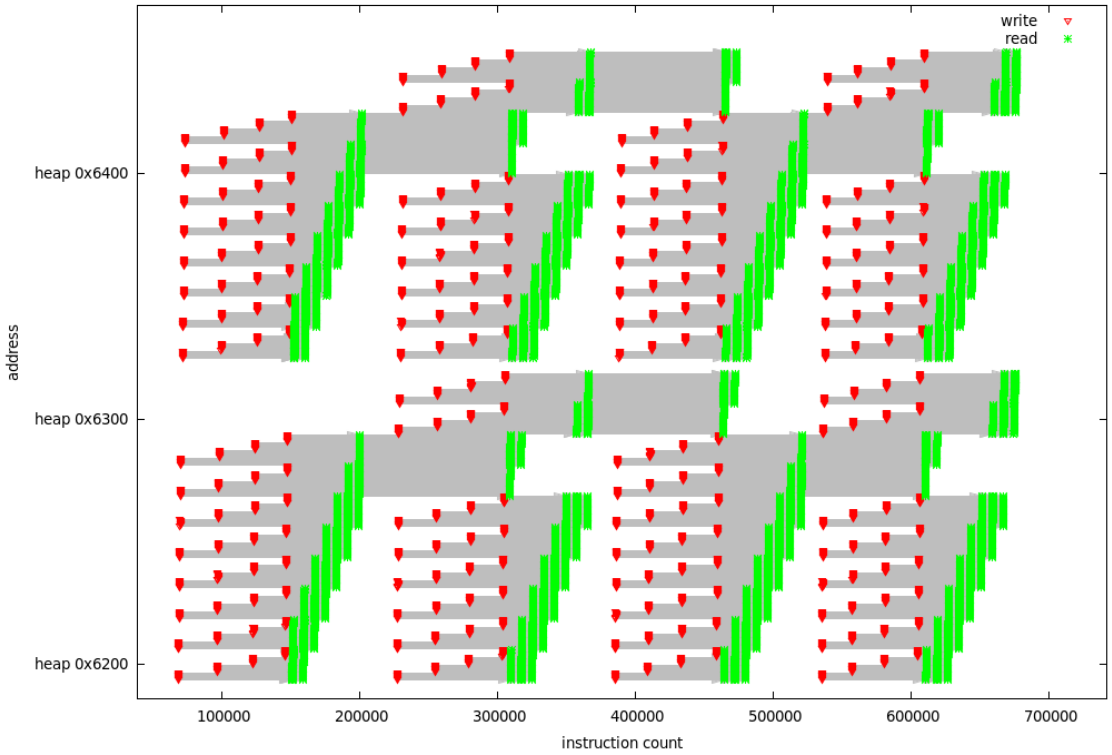
6.12. ábra. *Az inverz DCT által felhasznált együttható blokkok azonosítása*

Miután az entrópia dekódolás és inverz DCT műveletek eredményeként előállt egy MCU sor (jelen esetben négy MCU egymás mellett), a további feldolgozási lépések már pixel sorokat kapnak bemenetként. A következő lépéseket a 6.13. ábrán figyelhetjük meg.

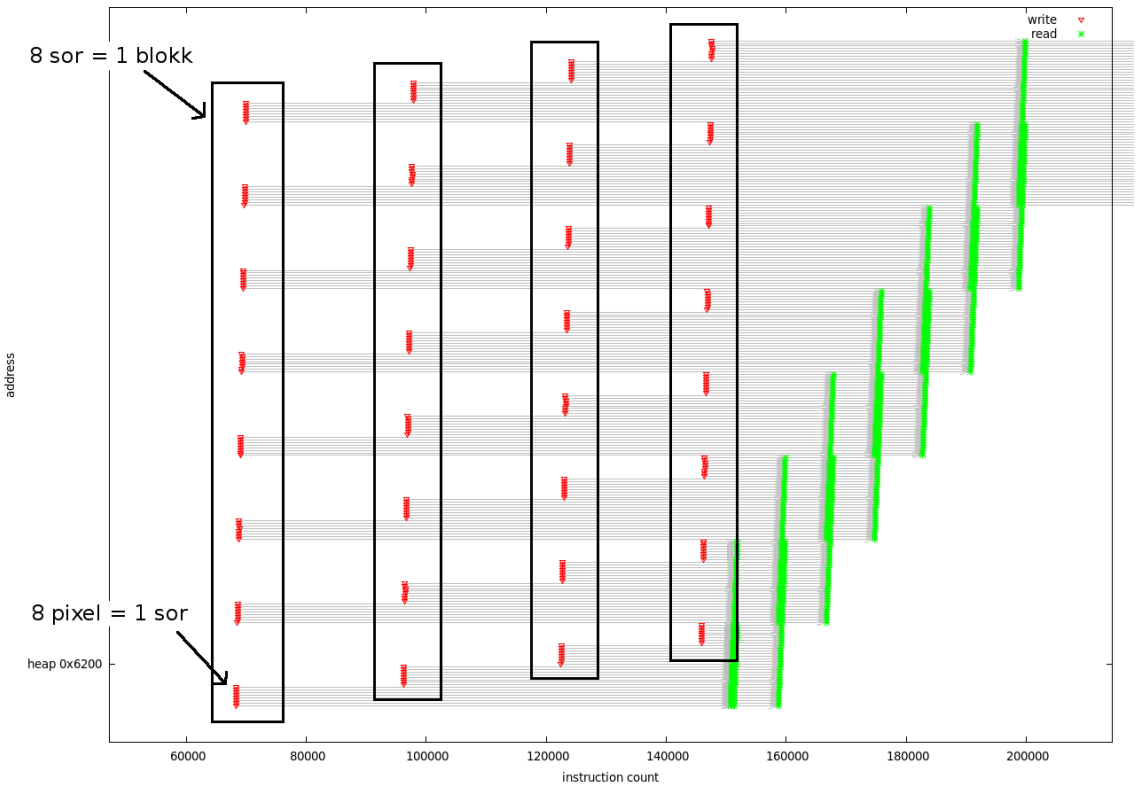


6.13. ábra. *Színkomponensek interpolálása és színtér transzformáció*

A `process_data_context_main` által meghívott `h2v2_fancy_upsample` függvény feladata a két színkülönbségi csatorna eredeti felbontásra történő interpolálása. A 6.14. ábrán a `jpeg_idct_islow` és a `h2v2_fancy_upsample` függvények közötti adatáramlás memóriacím és idő szerinti megjelenítése látható. A két színkülönbségi csatorna adatai jól láthatóan elkülönülnek a memóriacím szerint (az ábra alsó és felső fele). A 6.15. ábra az egyik csatorna első írási és olvasási műveleteit mutatja kinagyítva. Az írási műveleteket összevetve a `jpeg_idct_islow` függvény forráskódjával, értelmezhetjük a látottakat. A transzformáció természetesen 8×8 pixeles blokkokban történik, és az eredmény pixelek soronként íródnak a kimeneti bufferbe. Az ábrán a soroknak az írási műveletek kis függőleges csoportjai felelnek meg. Nyolc ilyen sorból áll össze egy teljes blokk. Megfigyelhető, hogy a blokkok

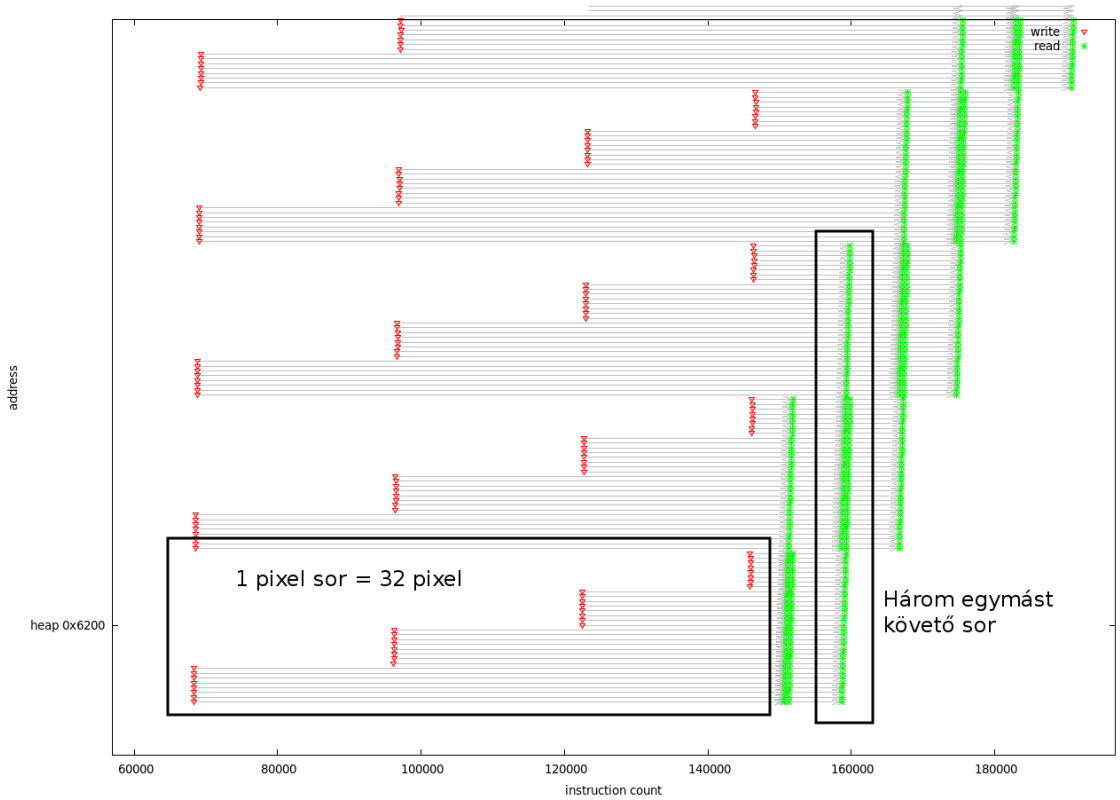


6.14. ábra. *Az inverz DCT-t és az interpolációt végző függvények közötti adatáramlás*



6.15. ábra. *Az inverz DCT függvény kimeneti adatfolyamának magyarázata*

egymás alatti sorai a bufferben nem közvetlenül egymás mellett helyezkednek el. Ezzel a megoldással biztosítható, hogy a további blokkok transzformálása után a kép teljes sorai folytonosan álljanak rendelkezésre a memóriában.

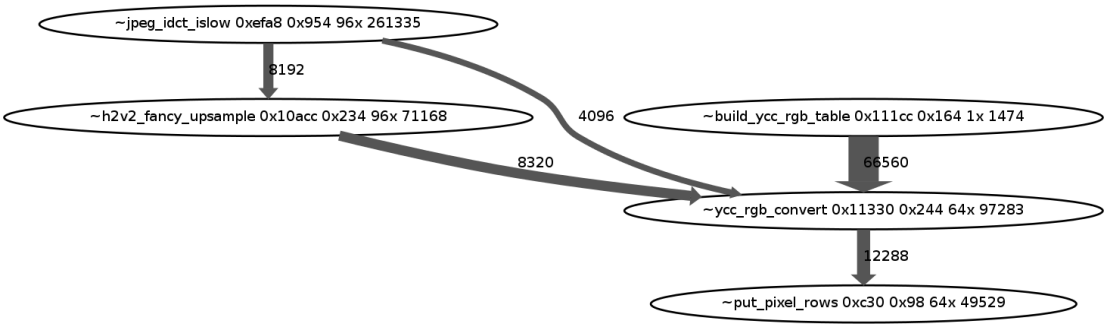


6.16. ábra. *Adatok felhasználása a színkomponens interpolálásához*

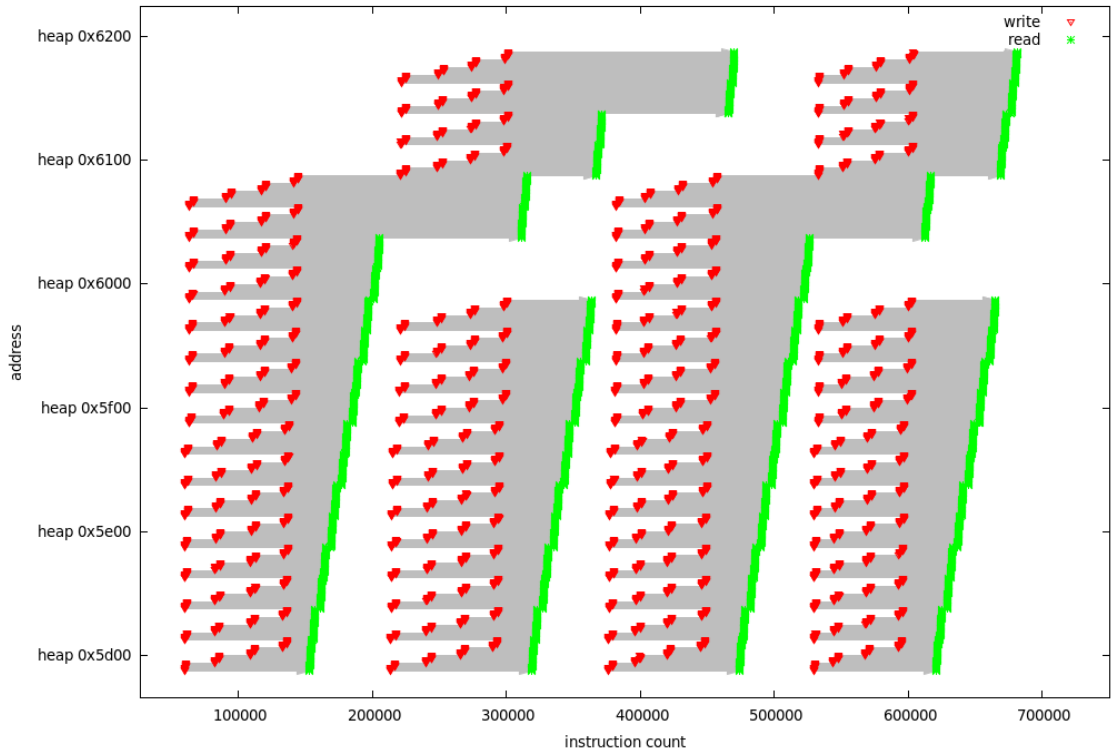
A 6.16. ábrán az interpolálás működésébe nyerhetünk bepillantást az olvasási műveletek értelmezésével. Az interpolálás során egy menetben két kimeneti sor keletkezik, kétszeres vízszintes felbontással. Ehhez az alkalmazott lineáris interpoláció mind függőleges, mind pedig vízszintes irányban felhasználja a szomszédos pixeleket. Ennek megfelelően minden menetben három egymást követő teljes sor beolvasása szükséges, ahogy az az ábrán is jól látható.

Ezzel kapcsolatban egy további érdekes megoldás is megfigyelhető a 6.14. ábrán. Látható, hogy a négy MCU sor feldolgozása közben az MCU sorok utolsó két pixel-sora nem kerül felülírásra a bufferben. Ez a megoldás biztosítja, hogy minden sor interpolálásához elérhető legyen a szükséges kontextus, azaz a megfelelő szomszédos sorok.

A dekódolás utolsó lépése a színtér konverzió, amit az `ycc_rgb_convert` függvény hajt végre. A 6.17. ábrán az aggregált adatfolyam gráf idevágó részlete figyelhető meg. Ahogy látható, az `ycc_rgb_convert` függvény adatokat fogad közvetlenül az inverz koszinusz transzformációt végző `jpeg_idct_islow`, valamint a színkülönbségi csatornák interpolálását végző `h2v2_fancy_upsample` felől. A feldolgozás itt is pixel-soronként történik, amelyek immár azonos felbontásúak (64 pixel).

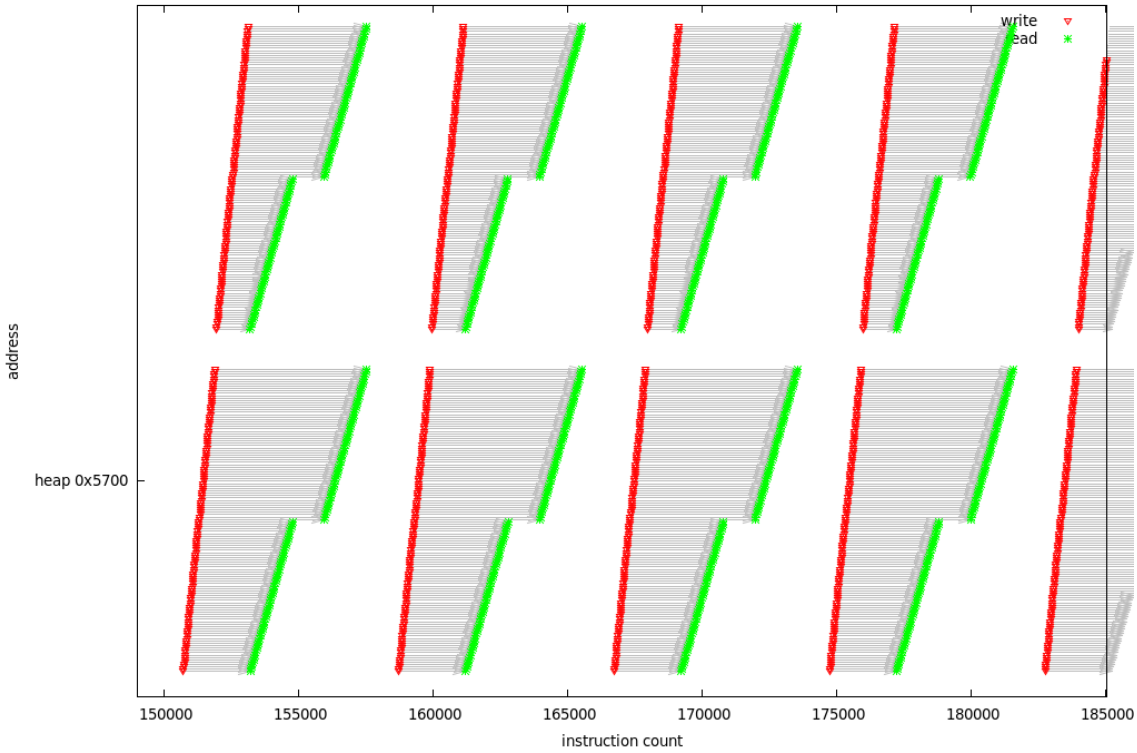


6.17. ábra. *Színtér konverzió függvényei*



6.18. ábra. *Színtér konverzió bemenete az inverz DCT felől*

A 6.18. ábrán a `jpeg_idct_islow` és az `ycc_rgb_convert` függvények közötti adatáramlás idő és memóriacím szerinti megjelenítése látható. Vegyük észre, hogy az ábrázolt adatáramlás nagyban emlékeztet a `jpeg_idct_islow` és a `h2v2_fancy_upsample` függvények között látottakra (6.14. ábra). Ennek oka természetesen, hogy az adatok forrása mindkét esetben az inverz DCT transzformáció. A két ábra között azonban eltéréseket is felfedezhetünk, amelyek alapvetően abból adódnak, hogy a fényesség csatorna esetén minden MCU négy DCT blokkot tartalmaz. A 6.18. ábrán – a 6.14. ábrához hasonlóan – szintén megfigyelhető, hogy a buffer utolsó sorai nem íródnak felül az új MCU sor feldolgozásakor. Ez biztosítja, hogy a fényességi csatorna sorai az interpolált színkülönbségi csatornák soraival szinkronban legyenek elérhetőek a színtér transzformáció elvégzéséhez.



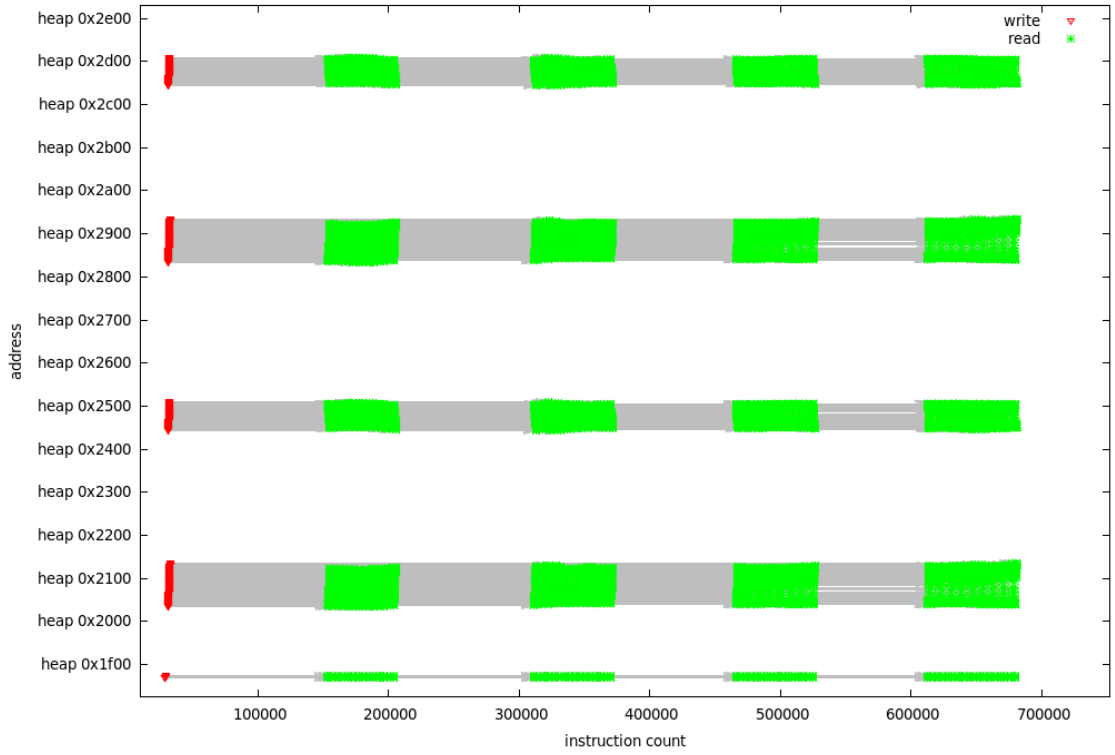
6.19. ábra. *Színtér konverzió bemenete az interpoláló egység felől (részlet)*

A 6.19. ábrán a `h2v2_fancy_upsample` és az `ycc_rgb_convert` függvények közötti adatáramlás részlete látható. Megfigyelhető, a két színkülönbségi csatorna adatainak elkülönülése, valamint hogy az interpolálás egy menetben valóban két teljes sornak megfelelő (azaz 128 bájt) adatot ír a bufferbe. Ezzel szoros összefüggésben, az olvasási műveletek szakaszaiban látható „törések" azt mutatják, hogy a színtér konverzió csak soronként dolgozza fel a bufferbe írt adatokat.

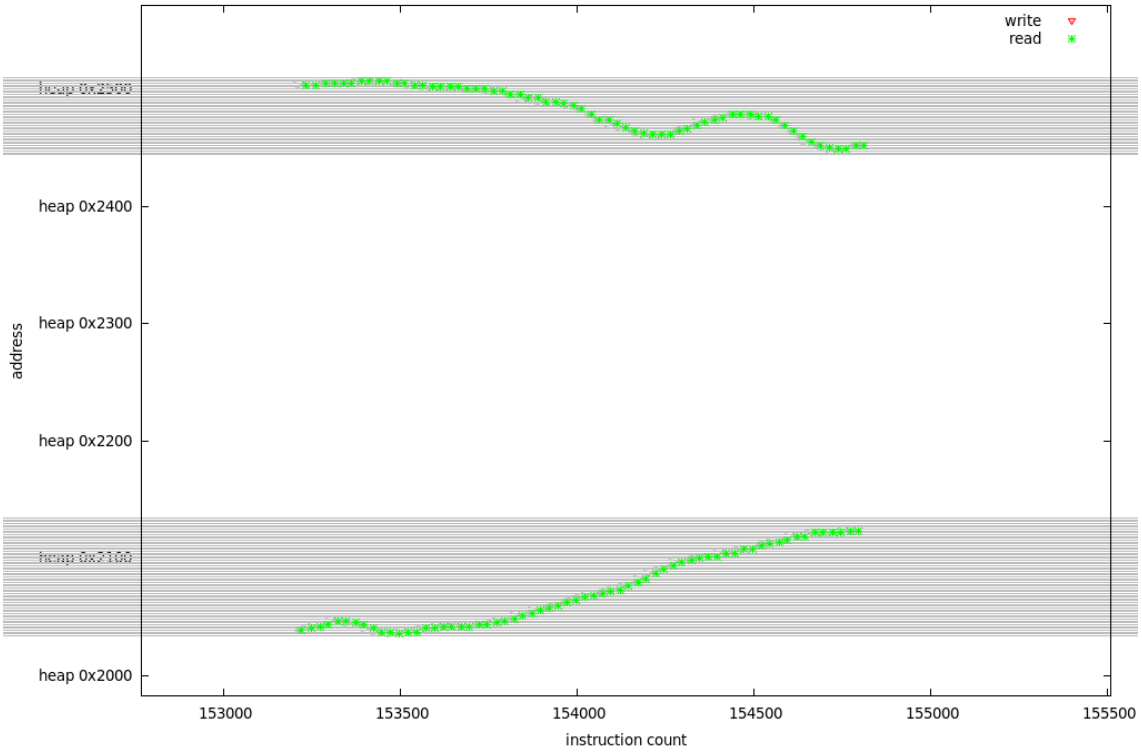
Az `ycc_rgb_convert` függvény egy további bemenete (adatfolyam szempontból) is megfigyelhető a 6.17. ábrán. Az inicializálás során már megemlített `build_ycc_rgb_table` függvény a szorzási műveletek gyorsítására szolgáló táblázatokat állítja elő. Ezek a táblázatok a konverzió elvégzéséhez szükséges négyféle konstans értékkel történő szorzások eredményeit tárolják az összes lehetséges 8 bites bemenet esetére.

A táblázatok előállításának és felhasználásának idő és memóriacím szerinti jellemzőit a 6.20. ábra mutatja. A memóriacím szerint megfigyelhető tagolódás oka, hogy miután a színkülönbségi csatornák pixel-értékei között a teljes skála csak egy kis tartományába eső értékek szerepelnek, az így nem használt táblázat elemekhez tartozó műveletek nem jelennek meg.

A 6.21. ábrán a szorzási táblázat elemeire hivatkozó olvasási műveletek láthatók kinagyítva, egy pixel-sorra és két táblázatra vonatkozóan. Megfigyelhető a sorban található pixelek értékeinek megfelelő táblázat elemek kiolvasása.

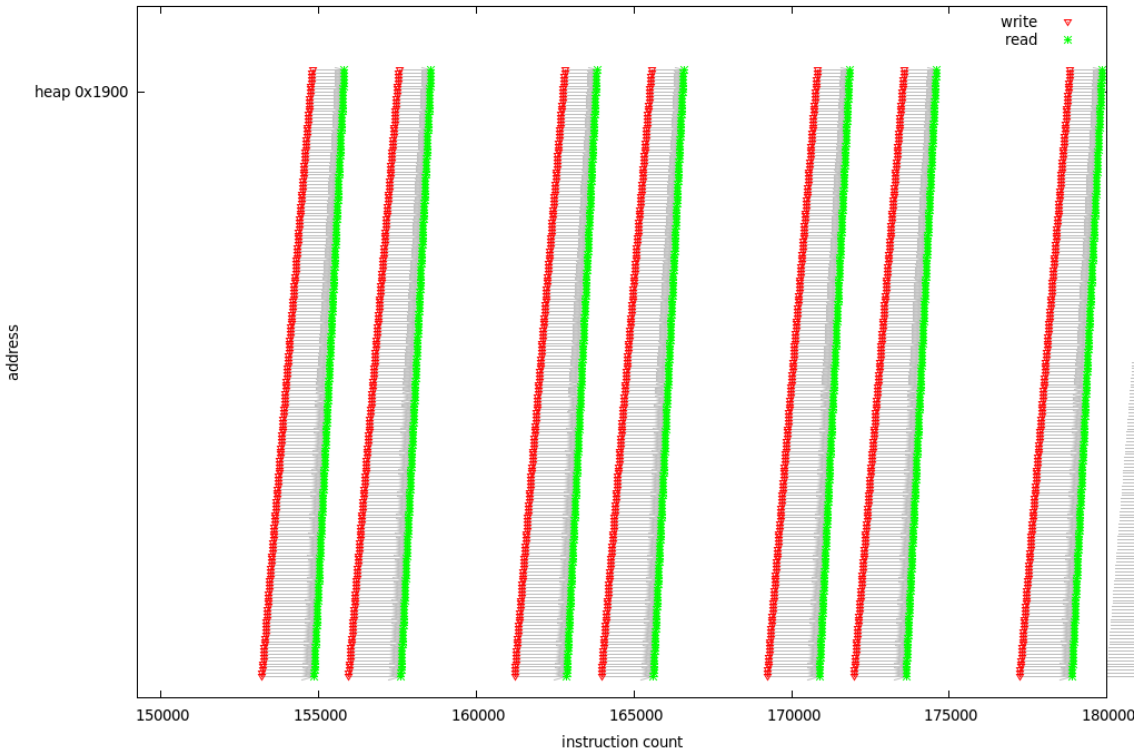


6.20. ábra. *Színtér konverzió szorzás gyorsító táblák előállítása és felhasználása*



6.21. ábra. *Színtér konverzió szorzás gyorsító táblák előállítása és felhasználása (részlet)*

Ahogy a 6.17. ábrán is látható, az `ycc_rgb_convert` függvény az RGB színtérbe átalakított pixel-sorokat a `put_pixel_rows` felé továbbítja. Az adatok a kimeneti fájlt író modul sorbufferében kerülnek átadásra, már közvetlenül a PPM fájlba írható formátumban. Az itt tapasztalható írási és olvasási műveletek egy kinagyított részletét a 6.22. ábra mutatja. Megfigyelhető, hogy a sorbuffer teljes írását egy teljes olvasási sorozat követi, majd a folyamat megismétlődik a kép összes sorára.



6.22. ábra. *Színtér konverzió kimenete a* put_pixel_rows *felé (részlet)*

A fentiek alapján egyértelműen belátható, hogy a vizsgált igen komplex JPEG megvalósítás megértését nagymértékben megkönnyítette az aggregált adatfolyam gráfok használata, amely sok – pusztán a forráskód alapján csak igen nagy energiabefektetés árán megérthető – részletre világít rá. Ha az adatfolyam gráfokat és a hozzá kapcsolódó egyéb ábrázolásokat a forráskód vizsgálatának kiegészítéseként használjuk fel, az nagyban megkönnyítheti a működés megértését. A 6.14. ábra esetén például a buffer kezelésénél tapasztalható sajátos megvalósítás megértését az adatfolyam memóriacím és idő szerinti ábrázolása nagyon szemléletessé teszi. Így a forráskód értelmezése is sokkal könnyebb lehet.

Ahogy tehát ebben a fejezetben láttuk, a létrehozott aggregált adatfolyam gráfok módszere jelen formájában is igen hatékony, azonban a továbbfejlesztési lehetőségeket bemutató fejezetben leírt bővített funkciók, kiegészítések és módosítások által még hatékonyabbá tehető.

6.4. Megjegyzések a párhuzamosíthatóságra vonatkozóan

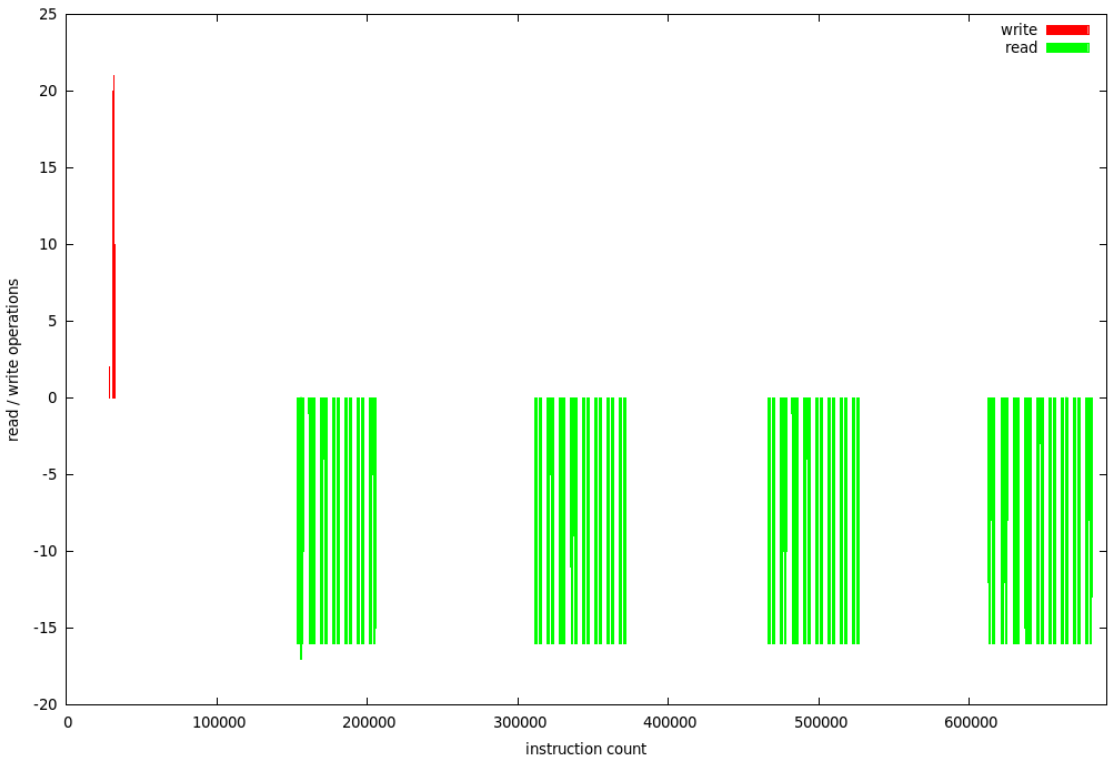
Ebben az alfejezetben az algoritmus vizsgálata során összegyűjtött tapasztalatok alapján megpróbálom csoportosítani a látott tipikus adatfolyamokat, majd röviden elemzem a különböző típusok párhuzamosíthatóságra gyakorolt hatásait. A vizsgált algoritmus alapján természetesen nem lehetséges teljesen általános érvényű kijelentéseket tenni, de a következőkben olvasható megállapítások nagy valószínűséggel általánosíthatók lesznek a továbbiakban.

A következőkben bemutatok három jól megkülönböztethető adatfolyam típust, amelyekre a JPEG algoritmus függvény szintű vizsgálata során példákat találtam. Nem jelenthető ki egyértelműen, hogy a függvény szintű vizsgálódás az ideális a párhuzamosíthatósági lehetőségek kereséséhez, azonban ennek megállapítása túlmutat a jelen dolgozat keretein.

Az adatfolyam típusok jellemzésére jól használható eszköznek tűnik az írási és olvasási műveletek idő szerinti eloszlásának megjelenítése, amely lényegében az eddig látott időbeli és memóriacím szerinti ábrázolás pontjainak idő szerinti összesítése által alakul ki. Így a vízszintes tengelyen a program futásának ideje, míg a függőleges tengelyen az adott időpontokban (illetve tartományokban) elvégzett műveletek száma látható.

Táblázat jellegű adatfolyam

A táblázat jellegű adatfolyamok jellemzője, hogy az adott memóriaterületekre egy írási művelet után nagy számú olvasás hajtódik végre. Az írási művelet jellemzően a program futásának elején, míg az olvasások a futás során elszórva találhatók.



6.23. ábra. *Táblázat jellegű adatfolyam példa*

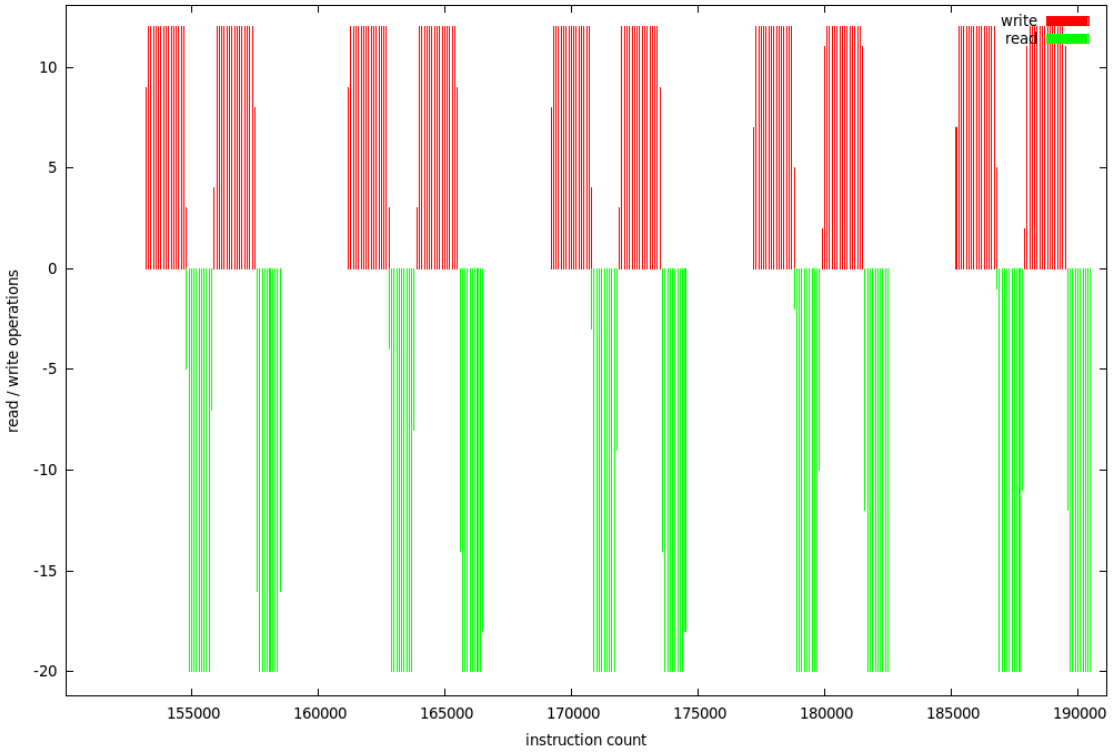
A vizsgált JPEG megvalósításban ilyen típusú adatfolyam figyelhető meg például a színtér konverziónál használt szorzási műveleteket gyorsító táblák esetén. Az adatfolyam a 6.20. ábrán látható. A 6.23. ábrán a műveletek idő szerinti összesítését láthatjuk. Jól megfigyelhető, hogy a futás elején található írási műveletek (pozitív tartomány) után jelentős mennyiségű olvasás (negatív értékek) következnek.

Az ilyen jellegű adatfolyamok alapvetően nem jelentenek szigorúan vett függőségi kapcsolatot az adatokat létrehozó és felhasználó függvények között. Ennek oka többek között az is, hogy a létrehozott adatok a futás teljes ideje alatt a bufferben maradnak, nem íródnak felül. A párhuzamosítás szempontjából a táblázat adatait a futás elején természetesen azok felhasználása előtt kell létrehozni, azonban ezután már nincs semmilyen függés.

Sorbuffer jellegű adatfolyam

A sorbuffer (esetleg nevezhetjük stream típusúnak is) jellegű adatfolyamok jellemzője, hogy az adott memóriaterületet a program működése során két függvény felváltva éri el. Az egyik függvény adatokat ír, míg a másik kiolvassa azokat.

Ilyen jellegű adatfolyam látható például a 6.22. ábrán. Ebben az esetben az adatokat az `ycc_rgb_convert` függvény szolgáltatja, és a `put_pixel_rows` függvény használja fel. Az adatáramlás idő szerinti összesítésének részlete a 6.24. ábrán figyelhető meg.



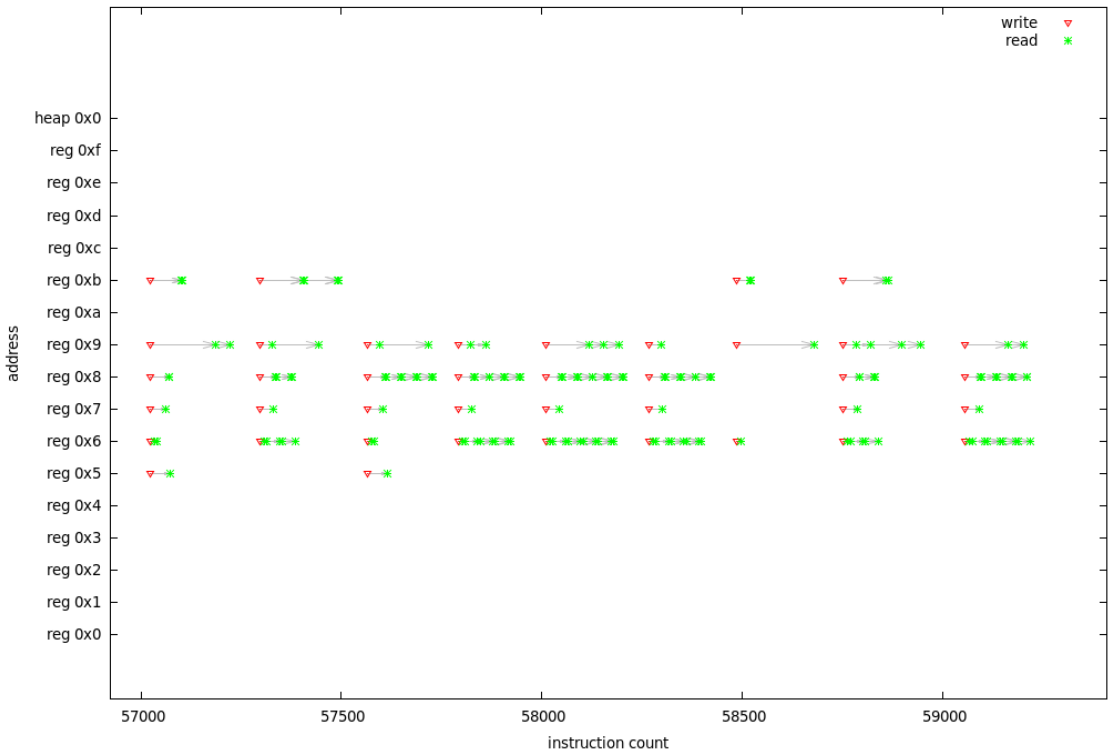
6.24. ábra. *Sorbuffer jellegű adatfolyam példa (részlet)*

Az ilyen jellegű adatfolyamok esetén a két függvény párhuzamosítása – ha más összefüggés nem áll fenn – elvégezhető. Mivel a párhuzamosan végrehajtott függvények alapvetően aszinkron jelleggel futnak (egymáshoz képest), valamilyen szinkroni-

zációs megoldás szükséges az időzítések biztosításához. Ez lehet egy szemafor jellegű eszköz, de például hardveres megvalósítás esetén használhatunk FIFO buffert is. A feldolgozási lépések párhuzamos végrehajtásával így egy *pipeline* jellegű struktúrát kaphatunk.

Függvényhívás jellegű adatfolyam

A függvényhívás jellegű adatfolyamok – ahogy az elnevezés is mutatja – függvényhívásokhoz kapcsolódnak. Az ilyen esetekben az adatok legtöbbször regiszterekben vagy a stack területen kerülnek átadásra. További lényeges jellemző, hogy a függvényhívás jellegű adatfolyamok csupán néhány adat írását és olvasását végzik alkalmanként, azonban gyakran ismétlődnek.



6.25. ábra. *Függvényhívás jellegű adatfolyam példa (részlet)*

A vizsgált program esetében ebbe a típusba sorolható például a `decode_mcu` és a `jpeg_fill_bit_buffer` közötti adatáramlás. A tényleges függvényhívásokat itt a `decode_mcu` hajtja végre. Az írási és olvasási műveletek idő és regiszter szerinti eloszlásának részlete a 6.25. ábrán figyelhető meg. Látható, hogy az írási műveleteket néhány olvasás követi, majd ismét írás következik. Érdemes megfigyelni, hogy amíg ebben az esetben az egyes írások között végrehajtott utasítások száma (eltelt idő) százas nagyságrendben van, a korábban látott sorbuffer jellegű adatfolyam írási blokkjai közötti távolság ennél lényegesen nagyobb volt.

Mivel ezen függvényhívások lényege, hogy a hívó függvénynek szüksége van a hívott függvény által szolgáltatott eredményre, a párhuzamosítás nem hajtható végre közvetlenül.

A bemutatott eseteken túl, sokféle kapcsolat típus fordulhat még elő a vizsgált algoritmusokban, amelyek mind eltérő módon befolyásolhatják a párhuzamosítás lehetőségeit. A feladat megoldásához nem elegendő egyes kapcsolatokat elemezni, azok teljes hálózatát kell vizsgálni. A létrehozott analízis eszköz – következő fejezetben leírt – továbbfejlesztési lehetőségei ennek elvégzéséhez is jó alapot biztosíthatnak.

7. fejezet

Összefoglalás

A diplomatervezés két féléve során feldolgoztam a hagyományos és memória profiling témakörében elérhető szakirodalmat, majd azok alapelveit felhasználva kidolgoztam az adatfolyam gráfok aggregálásának módszerét. Ez a módszer alkalmas a programok függvényei közötti kommunikáció elemzésére, több szempont szerint is. Az aggregált adatfolyam gráf megjelenítése mellett a kommunikáció időbeli és „térbeli" jellemzőinek ábrázolására is van lehetőség.

Az aggregált adatfolyam gráfokat létrehozó analízis eszköz valós helyzetben történő kipróbálásához egy JPEG dekódoló programot használtam. A program vizsgálata során megtapasztaltam, hogy az aggregált adatfolyamok milyen hatékonyan használhatók, elsődlegesen az algoritmusok működésének szemléltetéséhez és megértéséhez. Az elvégzett vizsgálat alapján ez az eszköz a párhuzamosíthatóságot akadályozó függőségek feltárásánál is hatékony segítség lehet. Ennek alátámasztásaként a dolgozat végén kiemeltem néhány megkülönböztethető adatfolyam típust, és röviden elemeztem azokat a párhuzamosíthatóság szempontjából.

Az elkészített eszköz tehát teljesíti a kitűzött követelményeket, és a tapasztalatok alapján a programok elemzéséhez nagyon hatékony eszközöket biztosít. Már a jelenlegi állapotban is látható, hogy a párhuzamosíthatóság vizsgálata során is jól hasznosíthatóak az aggregált adatfolyam gráfok, azonban a következőkben bemutatott továbbfejlesztési lehetőségek implementálásával ez még nagymértékben javítható.

7.1. További feladatok

A további fejlesztési feladatokat alapvetően két csoportra lehet osztani. Az elkészített analízis eszköz módosításához és bővítéséhez kapcsolódó feladatok alkotják az egyik csoportot, míg a másik csoportba a távolabbi jövőben megvalósítandó egyéb fejlesztési feladatok tartoznak.

Az analízis eszköz használata közben felmerült új funkciókat és lehetséges módosításokat, amelyek a párhuzamosíthatósággal kapcsolatos vizsgálatokat is még hatékonyabbá tehetik, a következőkben foglalom össze:

Változók nevének megjelenítése

A programok analízise során nagy segítséget jelentene, ha a memóriaműveletek megjelenítése nem csak a műveletek címeit mutatná, de az adatokat tároló változók neveit is. Ezzel még gyorsabban össze lehetne kapcsolni a megjelenített adatfolyamokat a vizsgált program forráskódjával.

A megvalósítás szempontjából eltérő nehézséget jelentenek például a globális változók, a regiszterekben vagy a stack-en lévő paraméterek vagy a mutatókon keresztül elért, dinamikusan allokált területek.

Egyedi adatok követése

A JPEG dekódoló program vizsgálata közben több alkalommal is nagyon hasznos lett volna egy (esetleg néhány) egyedi adat forrásainak függvényeken átívelő követése és megjelenítése. Itt tulajdonképpen az adott adatból indulva bejárható részgráf megjelenítése a feladat. Fontos megjegyezni, hogy az ehhez szükséges információ rendelkezésre áll, csupán a megjelenítést kell megvalósítani.

További érdekes összefüggéseket tárhatunk fel, ha egy kijelölt adat felhasználási pontjait jelenítjük meg hasonló módon. Ez tulajdonképpen a források ábrázolásának megfordításaként is felfogható.

Nem grafikus jellegű statisztikák készítése

Az aggregált adatfolyam gráfok élein és csomópontjain megjelenített néhány összesített érték mellett még sok más jellemző is hasznos információkkal szolgálhat, amelyeket nem a gráf elemein, hanem akár szöveges vagy táblázatos formában lehet megjeleníteni. Csomópontok (azaz függvények) esetén ilyen lehet például a különféle típusú (adatmozgató, aritmetikai, egyéb) utasítások összesítése, illetve a más függvényektől fogadott vagy azok számára előállított adatok számának listázása (azaz lényegében a csatlakozó élek felsorolása). Élek esetén hasonlóan érdekes lehet az adatokat felhasználó és előállító utasítások felsorolása, amely megmutathatja például, hogy egy függvény a bemeneti adatokat egy vagy több ponton fogadja-e.

A következőkben az analízis eszköz jelenlegi megvalósításától független, nagyobb feladatokat mutatom be röviden:

Teljesítmény növelése

Az analízis eszköz jelenlegi megvalósításában csak kisebb méretű programok vizsgálatára alkalmas. A feldolgozás ideje mellett a felhasznált nagy mennyiségű memória is erősen korlátozza az eszköz által vizsgálható algoritmusok körét. (A 64×64 pixeles JPEG fájl dekódolása alatt rögzített hozzávetőleg 700000 utasítás feldolgozásához 6 perc és 1.9GB memória szükséges.)

Valószínű, hogy az eszköz megvalósításához használt Python nyelv erősen hozzájárul ehhez a nagy erőforrásigényhez. A feldolgozási teljesítmény növeléséhez ezért szükséges lehet az eszköz valamilyen alacsonyabb szintű – például C++ – nyelven történő megvalósítása.

Vezérlési függőségek figyelembe vétele

A vezérlési függőségek – ahogy a 2. fejezetben bemutattam – a programban található feltételes elágazásokhoz köthetők. A párhuzamosíthatósági analízis szempontjából elengedhetetlen ezen függőségek figyelembe vétele. Ennek megvalósításához fel kell ismerni a feltételes utasításokat és az azok által a döntésekhez felhasznált adatokat. A függőségeket ezután a feltételes utasítások és az azok kimenetele által befolyásolt (lefuttatott vagy kihagyott) kódblokkok között állapíthatjuk meg.

Fordítás közbeni optimalizáció következményei

Az analízis eszköz fejlesztésének kezdetén megtapasztaltam (ezt dolgozatban nem tárgyaltam), hogy a vizsgált program adatfolyam gráfja jelentősen függ a GCC fordító esetén beállított optimalizációs szinttől. Érdekes feladat lehet ennek további vizsgálata, különös tekintettel a fordító által végzett függvény „behelyettesítésekre” és egyéb optimalizációs lépésekre.

Párhuzamosíthatóság vizsgálata gráfelméleti alapokon

Ahogy a 6.4. fejezetben is említettem, az algoritmus párhuzamosításának szempontjából az aggregált adatfolyam gráf egyes éleinek különálló vizsgálata nem elegendő. A teljes kép megalkotásához a függőségek hálózatát kell tekintenünk, amelyet ezután valamilyen gráfelméleti alapokon nyugvó módszerrel elemezhetünk.

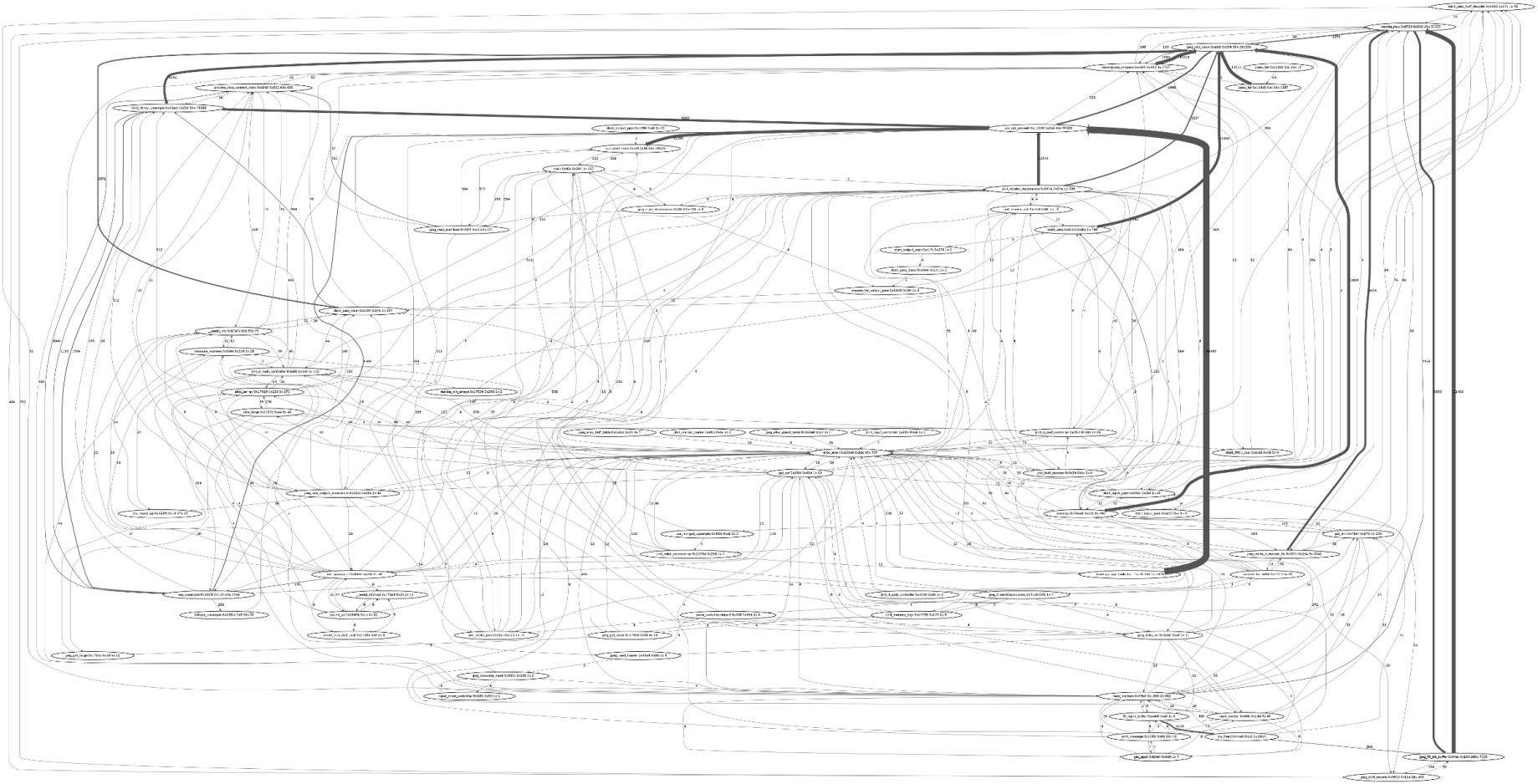
Köszönetnyilvánítás

Ezúton szeretném megköszönni konzulensemnek, Lazányi Jánosnak, hogy a feladatok megoldása során mindig szívesen fogadott, ha problémába ütköztem, és tanácsaival segítette előrehaladásomat.

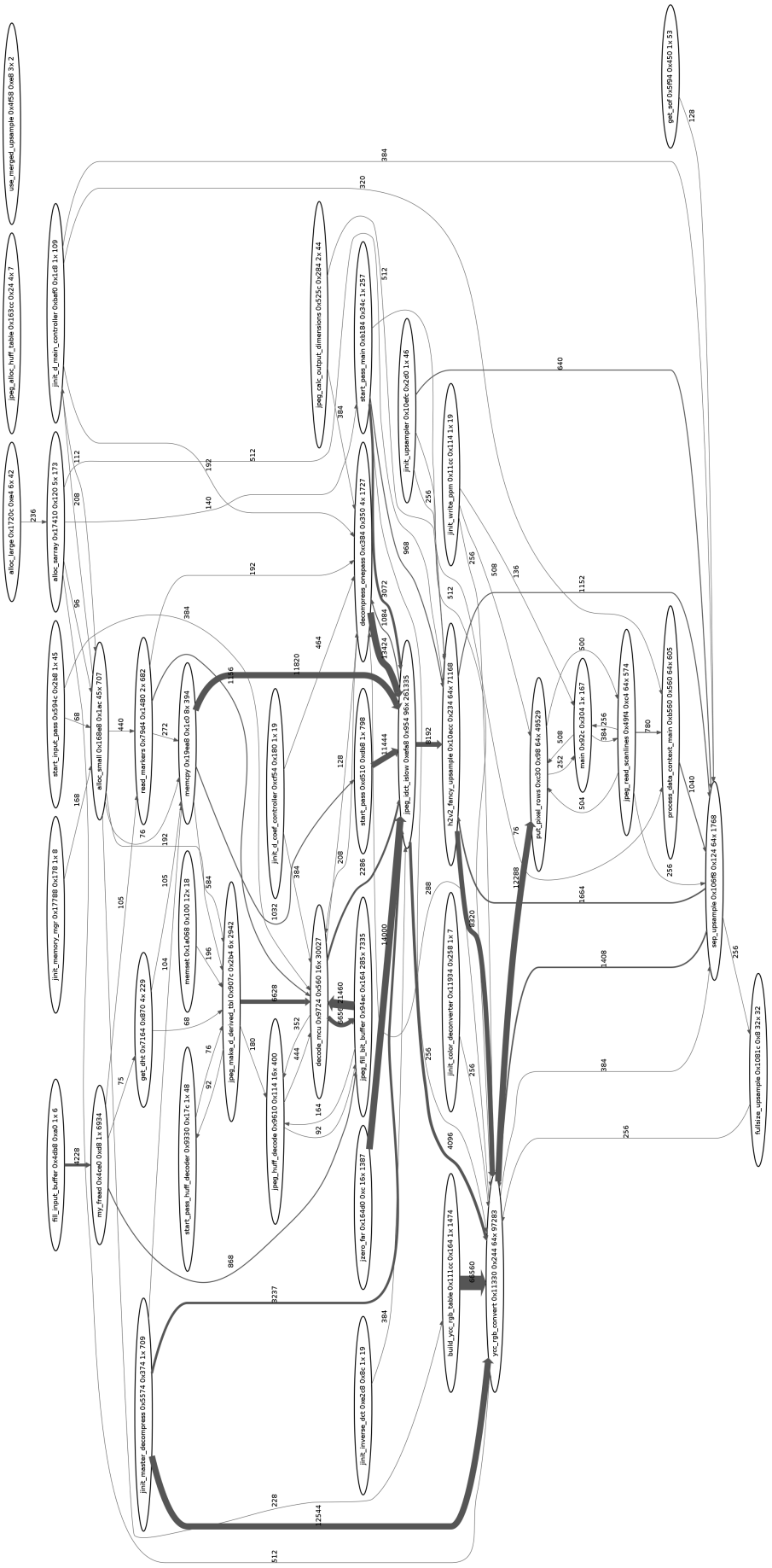
Függelék

F.1. Aggregált dinamikus adatfolyam gráf éleinek elhagyása

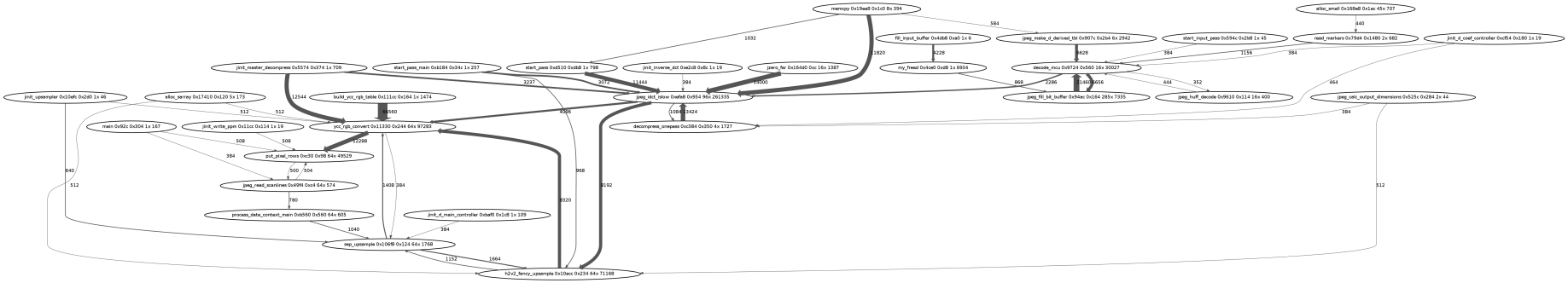
A következő ábrákon egy valós algoritmus aggregált adatfolyam gráfjai láthatók, különböző összesített adatáramlási határértékek alkalmazásával. Az F.1.1. ábra a teljes gráfot mutatja, élek elhagyása nélkül. Az erősen lecsökkentett felbontás ellenére is megfigyelhető, hogy a nagy mennyiségű él igen áttekinthetetlenné teszi az ábrázolást. Az F.1.2. ábrán látható gráfból a legnagyobb összesített adatfolyam értékkel rendelkező élhez képest a 0.1% alatti éleket hagytam el. (Ez 230 él elhagyását jelenti.) Az így módosított gráf már sokkal áttekinthetőbb, jobban követhető. Az F.1.3. és az F.1.4. ábrák a további élek elhagyásának hatásait mutatják. Az így keletkező ábrázolások egyre inkább a vizsgált program *lényegét* ragadják meg. Ahogy látható, nehéz megállapítani az optimális határértéket úgy, hogy a program működése (és párhuzamosíthatósága) szempontjából ne hagyjunk el fontos élet.



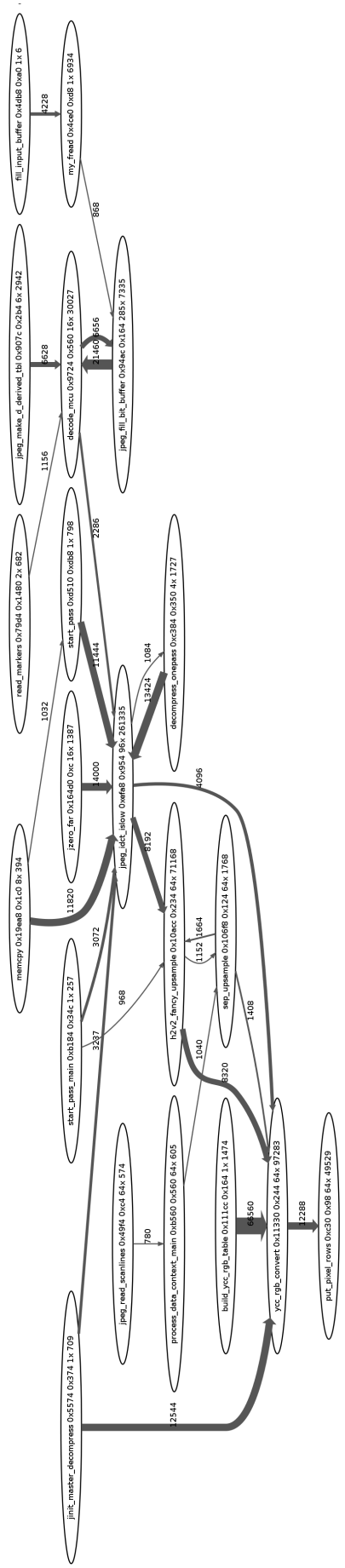
F.1.1. ábra. *A teljes aggregált adatfolyam gráf*



F.1.2. ábra. *Aggregált adatfolyam gráf 0.1% határérték esetén*



F.1.3. ábra. *Aggregált adatfolyam gráf 0.5% határérték esetén*



F.1.4. ábra. *Aggregált adatfolyam gráf 1% határérték esetén*

Irodalomjegyzék

[1] S. L. Graham, P. B. Kessler, and M. K. McKusick, „gprof: a call graph execution profiler," in *SIGPLAN Symposium on Compiler Construction*, pp. 120–126, 1982.

[2] A. Srivastava and A. Eustace, „Atom - a system for building customized program analysis tools," in *PLDI*, pp. 196–205, 1994.

[3] „Valgrind user manual." `http://valgrind.org/docs/manual/mc-manual.html`.

[4] „Kprof." `http://kprof.sourceforge.net/`.

[5] „Xgprof – a faster gprof with a nice gui." `http://sed.free.fr/xgprof/`.

[6] N. Nethercote and J. Seward, „Valgrind: a framework for heavyweight dynamic binary instrumentation," in *PLDI* (J. Ferrante and K. S. McKinley, eds.), pp. 89–100, ACM, 2007.

[7] „Qemu – open source processor simulator." `http://wiki.qemu.org/Main_Page`.

[8] J. Seward and N. Nethercote, „Using valgrind to detect undefined value errors with bit-precision," in *USENIX Annual Technical Conference, General Track*, pp. 17–30, USENIX, 2005.

[9] N. Nethercote and J. Seward, „How to shadow every byte of memory used by a program," in *VEE* (C. Krintz, S. Hand, and D. Tarditi, eds.), pp. 65–74, ACM, 2007.

[10] J. Weidendorfer, M. Kowarschik, and C. Trinitis, „A tool suite for simulation based analysis of memory access behavior," in *International Conference on Computational Science* (M. Bubak, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, eds.), vol. 3038 of *Lecture Notes in Computer Science*, pp. 440–447, Springer, 2004.

[11] „Cachegrind: a cache and branch-prediction profiler." `http://valgrind.org/docs/manual/cg-manual.html`.

[12] N. Nethercote and A. Mycroft, „Redux: A dynamic dataflow tracer," *Electr. Notes Theor. Comput. Sci.*, vol. 89, no. 2, pp. 149–170, 2003.

[13] „Valgrind – variants and patches." `http://valgrind.org/downloads/variants.html?njn`.

[14] J. Mak and A. Mycroft, „Limits of parallelism using dynamic dependency graphs," 2009.

[15] T. M. Austin and G. S. Sohi, „Dynamic dependency analysis of ordinary programs," in *ISCA* (A. Gottlieb, ed.), pp. 342–351, ACM, 1992.

[16] A. Forin, B. Neekzad, and N. L. Lynch, „Giano: The two-headed system simulator," tech. rep., Microsoft Research, 2006.

[17] „Sourcery codebench portal." `http://sourcery.mentor.com/GNUToolchain/`.

[18] „Valgrind Technical Documentation – Callgrind Format Specification." `http://valgrind.org/docs/manual/cl-format.html`.

[19] „Graphviz – Graph Visualization Software." `http://www.graphviz.org/`.

[20] J. Fonseca, „XDot – Interactive viewer for Graphviz dot files." `http://code.google.com/p/jrfonseca/wiki/XDot`.

[21] „yEd Graph Editor." `http://www.yworks.com/en/products_yed_about.html`.

[22] „Python Programming Language – Official Website." `http://www.python.org/`.

[23] „Epydoc – Automatic API Documentation Generation for Python." `http://epydoc.sourceforge.net/`.

[24] „PyDev." `http://pydev.org/`.

[25] „git." `http://git-scm.com/`.

[26] „Independent JPEG Group." `http://www.ijg.org/`.

[27] ITU, *ISO/IEC 10918-1: Information technology – digital compression and coding of continuous-tone still images – requirements and guidelines.* 1993.

[28] ITU Radiocommunication Assembly, *Rec. ITU-R BT.601-4: Encoding parameters of digital television for studios.* 1994.

[29] N. Ahmed, T. Natarajan, and K. R. Rao, „Discrete cosine transform," *IEEE Transactions on Computers*, vol. C-32, pp. 90–93, 1974.