Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Irányítástechnika és Informatika Tanszék

Modellalapú új módszer heterogén többprocesszoros architektúrák rendszerszintű szintéziséhez

PhD Tézisfüzet

Suba Gergely

Témavezető:
Dr. Arató Péter
Professor emeritus, az MTA rendes tagja

Budapest, 2018

1. Bevezetés, előzmények

Napjainkban az FPGA (field-programmable gate array) és ASIC (application-specific integrated circuit) rendszerek tervezése leggyakrabban hardverleíró nyelveken (hardware description language, HDL) történik, ezek közül a legelterjedtebbek a Verilog, a VHDL és ezek továbbfejlesztett változatai. Ezek a nyelvek a digitális áramköröket modularizált módon felépíthető logikai műveletekkel, tárolókkal és a közöttük futó összeköttetésekkel (bitvezetékekkel) írják le. A HDL nyelvek fontos tulajdonsága, hogy az áramkör időzítési aspektusát is modellezik az órajelek explicit leírásával.

Egy következő lépésként a magasszintű szintézist (high-level synthesis, HLS) [NSP+16] mint egy újabb, absztrakciós szinteket áthidaló módszert vezették be. A HLS lényege, hogy a hardverleírást automatikusan lehet előállítani a hardveres fogalmaktól és időzítésektől mentes, tisztán algoritmikus feladatmegfogalmazás alapján. Az előállítás közben optimalizálási (szinkronizációs, ütemezési és allokációs) feladatok elvégzése indokolt a kimenetként szolgáló hardverleírás minőségének garantálása érdekében. Az 1990-es években a hardvertervezés általános eszköztáraként tekintettek a HLS-re [MPC90, Cam90], majd az FPGA-k elterjedésével egyre inkább fókuszba kerültek az FPGA alapú beágyazott rendszerek mint a HLS fontos használati területei [CGMT09, MST+09, GAGS09, CNNV11, CDL11, MVG+12, SL16].

A HLS használata előnyös a szoftverfejlesztő mérnökök számára, mert segítségével az általuk megszokott fejlesztői nyelven a szoftvereknél hatékonyabban működő hardveres gyorsítókat, vagy akár teljes célhardvereket állíthatnak elő behatóbb villamosmérnöki ismeretek nélkül. A HLS szintén előnyös a hardverfejlesztő mérnököknek, mert a HDL-hez képest magasabb absztrakciós szintről kezdhetik a fejlesztést egy hardver tervezése és implementálása során.

A tervezési módszerek bemenő absztrakciós szintjét még magasabbra emelve jutottak el az elektronikus rendszerszintű (electronic system-level, ESL) tervezésig. ESL tervezést használva magasszintű viselkedési leírásokból kiindulva hardvert és szoftvert egyaránt tartalmazó rendszerek szisztematikus fejlesztésére nyílik lehetőség. Az automatizált ESL tervezési folyamatra ESL szintézisként, vagy csak egyszerűen rendszerszintű szintézisként (system-level synthesis, SLS) hivatkozik a szakirodalom [GHP+09].

A HLS és az SLS többféle területen is nagy segítséget nyújt. Egyrészt alkalmas digitális célhardverek fejlesztésekor prototípusok, vagy akár a termékek gyors és automatizált megvalósítására, de ezenkívül speciális, pl. FPGA-t is tartalmazó szuperszámítógépek platformjára szakosodott fordítóként is szerepet kaphat. Ez utóbbi használati területet az újrakonfigurálható számítás (reconfigurable computing, RC) [CDW10, HDA08, TCW+05] és a nagyteljesítményű számítás (high-performance computing, HPC) [VN14, Awa09] fedi le.

A hardverleíró nyelvek (azaz a HLS szoftverek kimenetei) az adatfolyam nyelvek [WP94, NLG99, JHM04] családjába sorolhatóak. Ennek magyarázata az adatfolyam számítási modell és a digitális hardverek strukturális leírása közti analógia. A strukturális hardverleírás a műveletvégző modulok példányosításait és az azokat összekötő bitvezetékeket foglalja magában. Az előbbi az adatfolyam modellek műveletvégzőinek (folyamatok, aktorok), utóbbi az összekötő csatornáknak (folyamok, adatkapcsolatok) feleltethetőek meg.

A mai HLS fejlesztőeszközök legnagyobb része az imperatív és objektum-orientált nyelvcsaládokba tartozó C vagy C++ alapú bemeneti forráskódokat dolgozzák fel. Ezek között megtalálhatóak a legnagyobb EDA (electronic design automation) vállalatok termékei: Vivado HLS, CatapultC, CtoS, Synphony C, ill. akadémiai szoftverek is: LegUp, Bambu, GAUT stb. [NSP+16, AANS+14, MVG+12, CNNV11]. HLS bemeneti nyelvként léteznek – igaz, sokkal kisebb arányban – a szoftverfejlesztés területéről ismert nem imperatív nyelvek változatai is. Ilyenek pl. a funkcionális nyelvek családjába tartozó, Haskell[1] alapú Lava [BCSS98] és CλaSH[2] [BKK+10].

A HLS forrásnyelvek között a C nyelv kimagasló arányának oka az, hogy a legtöbb esetben a HLS olyan szakterületeken (hardveres rendszerek, alacsonyszintű szoftveres algoritmusok, beágyazott rendszerek) kerül alkalmazásra, ahol manapság nagyrészt a C alapú fejlesztés dominál. Ezen kívül nem elhanyagolható, hogy az informatika elmúlt évtizedeiből rengeteg algoritmus áll rendelkezésre C nyelven, amelyek hardverben történő alkalmazása C alapú HLS esetén viszonylag egyszerű (mivel nem szükséges a költséges, nyelvek közötti átalakítás).

Az imperatív paradigma nagymértékben eltér a hardverleíró nyelvektől. Egy hardverleírás egymással konkurens műveleteket, és az azokból kialakított modulokat foglal magában, míg egy imperatív nyelv szekvenciális kódot ír le. Ez az alapvető különbség az oka az imperatív nyelveket feldolgozó HLS-ek legnagyobb nehézségeinek [Edw05]. A következőkben sorra vesszük a három legnagyobb kihívást, amelyek az imperatív nyelvek HLS bemenetként történő alkalmazásánál kerül elő: globális változók, mutatók és vezérlési szerkezetek.

A magasszintű imperatív nyelvekben az állapot változók formájában jelenik meg. A változók értékét az utasítások megváltoztathatják, sőt, globális változók esetében ezt a program bármely pontján megtehetik. Hardverszintézis során ez különösen nagy problémát jelent, hiszen ilyenkor csak költséges buszrendszerekkel lehet biztosítani a közös memóriához való hozzáférést (az imperatív nyelvekhez közel álló processzoros rendszerekben is ezt teszik).

Az imperatív nyelvek jellemző elemei a mutatók (pointerek), amelyek a memó-

[1] `https://www.haskell.org`
[2] `http://www.clash-lang.org/`

ria egyes címeire mutatnak, és segítségükkel indirekt módon lehet a memória adott címeiről olvasni, vagy oda írni. HLS során a cél az, hogy minél inkább elosztott memóriával dolgozzunk (annál inkább lehet párhuzamosan futtatni a kódrészeket), ezt pedig negatívan befolyásolja, ha egy mutató szabadon hozzáférhet a teljes memória tartalmához.

Az imperatív programok vezérlési szerkezetei szintén nagy kihívást jelentenek. Egy egyszerű példa a nagyon gyakran használt return utasítás, amely egy függvény bármelyik pontján meghívható, és a függvényből való azonnali kilépést eredményez. Processzoros futtatás során ez teljesen természetes és könnyen implementálható, viszont párhuzamos feldolgozású hardverben (ill. adatfolyam modellben is) az ugró utasítások nem értelmezettek. Hasonlóan problémát jelent a ciklusok során alkalmazott break és continue, ill. a túl nagy szabadságot adó goto ugró utasítás. Ez utóbbit gyakorta tiltják a kódolási szabványok vagy ajánlások is.

Bár a funkcionális nyelvek HLS-ben manapság kevésbé elterjedtek, a szoftverfejlesztés területén használatuk egyre népszerűbb. Az elmúlt években megfigyelhető volt egy trend, miszerint a funkcionális nyelvi elemek (pl. lambda kifejezések, adat streamek, immutable változók) bekerültek a mai, széleskörűen elterjedt, alapvetően imperatív, ill. objektum-orientált nyelvekbe. Pl. C# a 3-as verziótól (2007), C++ a 11-es verziótól (2011), Java a 8-as verziótól (2014) támogatja a lambda kifejezéseket. A szoftverfejlesztés területén bevált nyelvi elemek viszonylag nagy késéssel ugyan, de megjelennek a hardverszintézis területén. Emiatt a funkcionális nyelvi elemek nem kerülhetnek ki egy korszerű HLS rendszer eszköztárából.

A funkcionális nyelvek a deklaratív nyelvek családjába tartoznak. Ezekre jellemző, hogy az utasítások végrehajtási sorrendjét nem adják meg explicit, szemben az imperatív nyelvekkel, ahol az utasítások végrehajtási sorrendjét a forráskód meghatározza. Ez azt is jelenti, hogy az imperatív nyelvekből ismert ugró utasítás és annak magasabb szintű megvalósulásai (elágazás, ciklus) nem írhatók le. Az elágazás helyett feltételes kifejezést, a ciklusok helyett pedig rekurziót alkalmaznak.

A funkcionális nyelvek másik fontos tulajdonsága a mellékhatásmentes működés. Tisztán funkcionális nyelvekben emiatt a változók csak egyszer kaphatnak értéket, ami az imperatív nyelvekhez képest egy nagyon fontos eltérés. Az imperatív nyelvekre jellemző, többször változtatható változókat az angol nyelvű szakirodalomban [But14] mutable, míg a funkcionális nyelvekre jellemző, egyszeri értékadású változókat immutable megnevezéssel látják el. Mivel a magyar terminológiában ezeknek nincs megfelelőjük, az előbbi fogalomra bevezetem az M-adat, míg az utóbbira az I-adat kifejezést.

A funkcionális nyelvek a mellékhatások tekintetében nagyon hasonlítanak az adatfolyam nyelvekhez. A tisztán funkcionális függvényre és az adatfolyam nyel-

vek műveleteire is igaz, hogy a kimeneti értékek csak a bemenetektől függhetnek (a szakirodalom ezt referencial transparency elvnek is nevezi [But14]). Ez a tulajdonság néhány tekintetben egyszerűbbé teszi a funkcionális nyelvről induló HLS-t az imperatív esettel összehasonlítva.

A funkcionális nyelvek bizonyos tulajdonságai viszont távol állnak az adatfolyam nyelvektől, így a hardverleírásoktól is. Ilyenek a rekurzív függvényhívások, a függvények első osztályú típusként való használata és a magasabb rendű függvények (amelyek paraméterként függvényeket várnak) is.

A programnyelvek fordító szoftverei gyakorta háromszintű architektúrára épülnek [Mer03, LA03, LA04]. A frontend feldolgozza a forrásnyelvet, és abból egy köztes reprezentációt állít elő. A middle-end a köztes reprezentáción optimalizálást végez, a backend pedig a köztes nyelvből előállítja a kimeneti tárgykódot. A háromszintű moduláris felépítés előnye, hogy egy újabb forrásnyelv kezeléséhez csupán egy új frontend modult kell beépíteni, a middle-end és a backend változtatás nélkül újrahasználható. Ehhez hasonlóan, egy újabb tárgykód előállításához csupán egy új backendet szükséges létrehozni. Az optimalizációs algoritmusokat pedig forrás- és tárgykódoktól függetlenül, a middle-endben szükséges csak implementálni.

Ezeknél a szoftvereknél a köztes reprezentáció nyelvének megválasztása az egyik legfontosabb architektúrális kérdés. Ettől függ, hogy a middle-end milyen optimalizációs algoritmusokat tud végrehajtani, ill. hogy a frontendnek és backendnek milyen átalakításokat kell elvégeznie a köztes nyelvre és a köztes nyelvről.

A ma létező HLS rendszerek tipikusan egy-egy forrásnyelvet, vagy egy adott paradigmának megfelelő, egymáshoz hasonló nyelveket kezelnek. Ennek oka, hogy az egymástól nagymértékben eltérő nyelvek (pl. imperatív vagy funkcionális) integrált kezelése bonyolult feladat, mivel azok fordítói teljesen eltérő köztes nyelvet használnak a forrásnyelvek alapvető különbségei miatt [Mer03, TC11].

2. Célok

Kutatásaim során többek között arra kerestem a választ, hogy a HLS rendszerekben milyen tulajdonságokkal rendelkező köztes nyelvet célszerű használni annak érdekében, hogy a szoftverfejlesztés területén bevált, háromrétegű architektúra előnyösen kialakítható legyen, és ezáltal egy HLS keretrendszer többféle forrásnyelvet is képes legyen feldolgozni, ill. többféle kimeneti nyelvet is képes legyen előállítani. A különféle forrásnyelvek HLS rendszerekbe történő egyszerű beépíthetősége napjainkban kifejezetten indokolt, mivel a szoftverfejlesztés területén nagyon sok forrásnyelv létezik, és a jövőben várhatóan újabb és újabb nyelvek fognak megjelenni, amelyek

előbb-utóbb a hardverszintézis területén is alkalmazásra kerülhetnek. (Megjegyzendő, hogy éppen az itt leírtak miatt, nem volt célom HLS-hez újabb forrásnyelv kidolgozása, a hardverszintézis során véleményem szerint érdemes a már létező, ismert és jól bevált programozási nyelveket használni, ez által lesz valóban széleskörűen és jól felhasználható a szintézisrendszer.) Az értekezés egyik célja tehát olyan új HLS köztes nyelv kidolgozása, amely figyelembe veszi az imperatív és a funkcionális paradigmák sajátosságait, de ezen kívül a HLS és az SLS fontos fázisai, a dekompozíció, az ütemezés és az allokáció is elvégezhető rajta.

Következő célként a HLS köztes nyelvet alapul vevő, imperatív és funkcionális nyelvű bemeneteket is feldolgozó fordító kidolgozása volt a célom. Legjobb tudomásom szerint nem létezik olyan HLS rendszer, amely többféle paradigma nyelveit (pl. imperatív és funkcionális) is kezelni tudja.

További cél a HLS köztes nyelv pipeline ütemezésének kidolgozása, amely a tervezett rendszer hatékonyságát nagymértékben növeli.

3. Új tudományos eredmények

A következő három alfejezetben az új tudományos eredményeimet ismertetem három tézis formájában.

3.1. Új köztes nyelv magasszintű- és rendszerszintű szintézishez

Elsőként sorra veszem azokat a nyelveket, amelyek potenciálisan felhasználhatóak a magasszintű- és rendszerszintű szintézis köztes nyelveként: az adatfolyamgráfokat, a szoftvermodellezési és általános gráf nyelveket, ill. az imperatív és funkcionális programozási nyelvek eszközkészletét.

Az adatfolyamgráfok tulajdonságai (mellékhatás-mentesség, adatkapcsolatok explicit jelölése), és a rendelkezésre álló ütemezési algoritmusok [Chi12, AVJ01, TC98] nagyon kedvezőek a hardverszintézis és a kimenetként előállítandó hardverleírások szempontjából. Viszont komoly hátrányokkal is rendelkeznek. Nem támogatják a hierarchikus gráfokat, így a forrásnyelvekre jellemző, strukturális és moduláris felépítést sem jelenítik meg. Az elágazásokat általában vezérlésfolyam (CFG) élekkel reprezentálják, így a gráf elveszti a tiszta adatfolyam jellegét. Az ilyen gráfokat szokás vezérlés- és adatfolyamgráfoknak (control and dataflow graph, CDFG) [NRE04] nevezni. CDFG-ben a ciklusokat sokszor csak visszairányú élek jelzik, így ezek a gráfban viszonylag költségesen, a körök keresésével találhatóak meg.

További hátrány, hogy az adatfolyamgráfok nem támogatják az M-adatokat. Ilyen esetekben Load és Store csomópontokat szokás bevezetni [SK86], amely azon-

ban szintén a tiszta adatfolyam jelleg elvesztésével jár. Továbbá, ezekben a gráfokban nem jelennek meg a komponensek és a komponenspéldányosítás fogalmak, azaz közös kódok (rutinok, függvények) írása általában nem támogatott.

A hátrányok miatt az adatfolyamgráfokat komplett HLS köztes nyelvként (IR, intermediate representation) alkalmazni nem lenne előnyös, de kijelenthető, hogy az új nyelv alapjait érdemes az adatfolyamgráf modellek ismereteire építeni.

Napjaink legelterjedtebb szoftvermodellezési nyelve az UML (Unified Modeling Language) [OMG15]. Az UML aktivitás diagram képes leírni adatfolyamgráfot, vezérlésfolyam gráfot vagy ezek bizonyos keverékét is (CDFG). Támogatja a hierarchikus felépítést. Ez nagy előny egy rendszer specifikálásánál, vagy az architektúra kialakításánál. Az UML aktivitás diagram egy általános, felhasználóknak szánt szoftvermodellezési eszköz, nem pedig egy, a HLS problémára szakosodott, fordítók belső reprezentációját megvalósító, alacsony elemszámú DSL.

Az általános gráf nyelvek (GraphML, GXL) [BEH$^+$, WKR02] jól illeszthetőek a gráfproblémákhoz, így bizonyos fokig egy adatfolyamgráfot is le tudnak írni. A hátrányuk viszont, hogy olyannyira általánosak, hogy a HLS szakterület elemei nem szerepelnek benne. Ennek ellenére, a nyelv bizonyos megoldásait (pl. hierarchia, hiperélek) érdemes figyelembe venni a HLS köztes nyelv kidolgozásánál.

A mai, egyik legszélesebb körben elterjedt imperatív fordító a GCC (GNU Compiler Collection) [EGH$^+$05], amely különböző forrásnyelveket és célplatformokat is támogat. Az ipari és kutatási projektekben is elterjedt másik népszerű fordítórendszer az LLVM [LA04], amely újragondolt architektúrával és külső nyelvként is használt köztes reprezentációval rendelkezik. Mindkét ismert fordítórendszer három rétegű (frontend, middle-end, backend), és köztes reprezentációjuk SSA (static single assignment) [CFR$^+$91, BP03] alapú.

Bár a skaláris változók ezekben a köztes nyelvekben valóban SSA formában adottak, a tömbökre és struktúrákra ez nem igaz, vagyis az adatstruktúrák SSA alapú modellezése nem megoldott. Hátrányuk, hogy a C nyelven még létező elágazások és ciklusok a reprezentációban már nem szerepelnek, ezeket a konstrukciókat ugró utasítások váltják fel. Az ugró utasításokból egyszerűbb esetekben lehetőség van visszanyerni az eredeti ciklusokat és feltételeket.

A GCC és LLVM IR-eket a hagyományos, processzorra szánt imperatív fordítók igényeire szabták, ezért érthető, hogy sokkal inkább hasonlítanak az assembly nyelvekre, mint egy adatfolyam leírásra. Jellemzőek bennük az adatfolyam megközelítéstől idegen, basic blockokat (BB) lezáró feltételes és feltétel nélküli ugró utasítások.

Az SSA forma jól illeszkedik az adatfolyamgráfokhoz, így ebből a szempontból a hardverszintézishez is előnyösen lehetne használni. A probléma viszont az, hogy az SSA forma csak skaláris változók esetében használatos, ill. ezek a nyelvek alapvetően

imperatívak (érthető okoknál fogva, hiszen processzoros rendszerekhez készültek), és így erősen építenek a vezérlésfolyam elvekre.

Az imperatív nyelvek eszköztárából a változók M-adatként történő kezelését érdemes az új nyelvbe is beépíteni, vannak ugyanis olyan problémák, amelyeknél a csupán I-adatokkal történő kezelés túlságosan nehézkes. Az imperatív nyelvek strukturális elemeit, az elágazásokat és ciklusokat szintén kezelhetővé kell tenni egy HLS IR-ben.

A funkcionális nyelvek köztes reprezentációi többnyire Lambda kalkulus alapúak. Ezekben a függvények első-osztályú típusok, azaz függvényekkel műveletek végezhetőek, és lehetőség van függvényeket más függvényeknek átadni (azaz lehetőség van magasabb-rendű függvények kezelésére). Ezek közvetlenül nyilvánvalóan nem ábrázolhatóak hardverleírásban, azaz egy lambda kalkulus alapú nyelvből ezeket először eliminálni kell. Ezt sok esetben béta redukcióval végzik, amely kódismétlést eredményezhet és az IR reprezentáció méretének növekedésével járhat. Az általam javasolt megoldás szerint a függvénytípusok és a hardverleírásban ábrázolható típusok is a nyelv elemkészletéhez tartoznak, de egymástól teljesen elválasztott formában (azaz a függvények ezután nem első-osztályú típusok lesznek). A funkcionális nyelvek további nehézsége az, hogy minden ciklussal megoldható problémát is rekurzióval oldanak meg. Hátrány az is, hogy M-adatokat alapvetően nem kezelnek.

Az általam javasolt HIG (HLS Intermediate Graph) adatfolyamgráf alapú, azaz a gráf csomópontok a műveleteket, az élek az adatkapcsolatokat reprezentálják. A műveletek előre definiált komponensek példányai. Egy komponens lehet elemi vagy összetett. Az elemi komponens tovább nem bontható műveletet ír le, az összetett komponens pedig adatfolyamgráfként adható meg, amely ezáltal hierarchikus felépítést eredményez. Az összetett komponensek között megjelenik a ciklus és az elágazás komponens (azaz a magasszintű strukturált nyelvek fontos építőelemei). A komponensek meghatározott számú ki- és bemeneti porttal rendelkeznek, amelyeket a példányosítás során a műveletek is örökölnek. Komponensek más komponenssel is paraméterezhetőek (hasonlóan, ahogy funkcionális nyelvekben függvények más függvényekkel paraméterezhetőek). A HIG az ütemezéshez szükséges információkat is képes modellezni függőségi és erőforrásélek formájában.

1. tézis. *Kidolgoztam egy új adatfolyamgráf alapú hierarchikus gráf modellt (HIG), amely az elektronikus rendszerszintű- és magasszintű szintézis köztes reprezentációjaként használható. A modell az eddig ismert adatfolyamgráfokkal szemben mind az I-adat (immutable, nem módosítható), mind az M-adat (mutable, azaz módosítható) szerinti feladatmegfogalmazások kezelését lehetővé teszi. Ezáltal egyaránt hatékonyan illeszthető a tisztán funkcionális és az imperatív paradigma szerinti forrásnyelvekhez is. Ellentétben a szokásos gráf modellekkel, az egymásba ágyazott ciklusokat és elágazásokat vezérlésfolyam nélkül ábrázolja. Ugyanakkor a hardverleírások elvonatkoztatási szintjén megjelenő komponensek, műveletek (azaz komponenspéldányok), portok, és az azokat összekötő adatkapcsolatok is részét képezik a HIG-nek. Modellezi továbbá a dekompozícióhoz, ütemezéshez és allokációhoz szükséges paramétereket is, így a teljes rendszerszintű szintézis folyamat során alkalmazható. A HIG-et formális nyelvtani leírással és a metamodell megadásával definiáltam. [1, 2, 3]*

3.2. Rendszerszintű szintézis funkcionális és imperatív nyelvekből kiindulva

A kereskedelmi és nyílt HLS rendszerekről a szakirodalomban időről-időre megjelennek összefoglaló tanulmányok [NSP+16, SL16, MVG+12, CDW10, MST+09]. A kereskedelmi szoftverek belső architektúrája a legtöbb esetben ipari titoknak minősül, így a továbbiakban a nyílt forráskódú szoftvereket elemzem, ezekről érthető okoknál fogva bővebb információ áll rendelkezésre. A tanulmányok alapján elmondható, hogy a legtöbb ma létező, nyílt forráskódú HLS és SLS rendszer szoftveres fordítórendszerek frontendjét használja, így pl. a BAMBU [PF13], a GAUT [CCB+08] és a HercuLeS [KM13] a GCC-t, a LegUp [CCA+11] és a Trident [TGP07] az LLVM-et, a CλaSH [BKK+10] a GHC-t.

Megfigyelhető, hogy ezek a szoftverek egy-egy nyelvi paradigmára adnak megoldást, így a GCC és az LLVM alapú megoldások imperatív nyelvekre (főként C alapúakra), a CλaSH a Haskell funkcionális nyelvre szakosodott. Ezzel szemben az értekezésben bemutatott módszerek integráltan kezelik az imperatív és a funkcionális bemeneti nyelveket.

Az I- és M-adatokat a rendszerszintű szintézis teljes folyamata során fontos kezelni.

Egyrészt, a forrásnyelvben bizonyos problémákat ésszerűbb I-adatként, míg másokat M-adatként leírni. Az előbbi pozitív tulajdonsága, hogy ilyenkor a változó csak egyszer kap értéket, így a fejlesztő vagy tesztelő számára egyszerűbben látható a forráskódból, hogy a program egy adott pontján lévő változó mely művelet eredményét használja fel. M-adat használata esetén az algoritmusok ciklussal is leírhatóak,

így nem szükséges ezeket rekurzióval kifejezni, ami sok esetben egyszerűbb, intuitív leírásmódot ad.

Másrészt, a szintetizált hardver hatékonyságát is nagymértékben befolyásolja az I-adat és M-adat közötti döntés. I-adat esetén a párhuzamosítás lehetősége megnő, hiszen ekkor csak az I-adatot író egyetlen csomópont és az azt felhasználó műveletek között áll fenn függés. Ezzel szemben M-adat esetén minden olyan csomópont függésben van, amely írja vagy olvassa az M-adatot, ami a párhuzamosítást általában negatívan befolyásolja. Ugyanakkor az M-adatnak hatalmas előnye van olyan algoritmusok implementálásakor, amelyben sokszor kell egy struktúra belső adatainak kisebb részleteit átírni (ez tömbök kezelésekor gyakorta előfordul). Ilyen esetekben az M-adat szerinti megoldás hatékonyabban kivitelezhető, mert nincs szükség folyamatos másolásokra.

Azt, hogy egy adatstruktúra I- vagy M-adatként kerül megvalósításra, a ma ismert megoldásokban már a forrásnyelv eleve meghatározza. Pl. C nyelven a tömbök, így a sztringek is M-adatok. Java nyelven a sztringek I-adatok, míg a tömbök M-adatok. Haskell nyelven a legtöbb adatstruktúra I-adat, de van lehetőség M-adatként is kezelni adatstruktúrákat (pl. MArray). Scala[3], Kotlin[4] vagy Xtend[5] nyelven a *var* kulcsszó M-adatot, míg a *val* kulcsszó I-adatot deklarál.

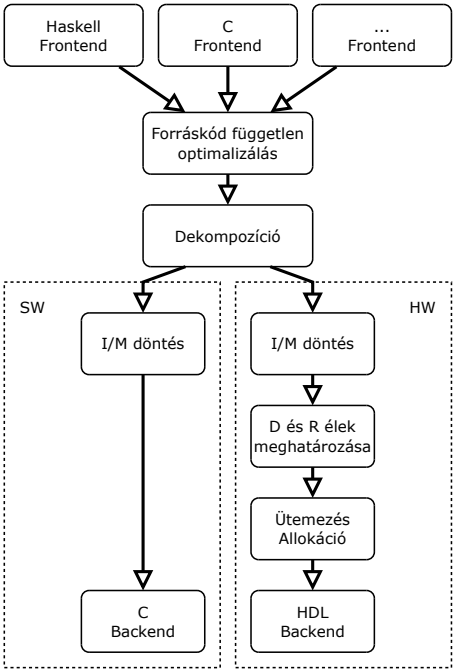
A jelenleg ismert fordítók megoldásaival a probléma az, hogy túlságosan hamar, már a forrásnyelvi leírásnál eldől, hogy a megvalósítás I- vagy M-adattal történik. Az általam javasolt megoldás szerint a forrásnyelv szintjén az I- vagy M-adat közötti döntés tetszőleges, így a felhasználó szabadon választhatja a nyelvhez és a problémához jobban illeszkedő leírásmódot. A megvalósítás során viszont a hardverszintézis körülményeihez (pl. idő és terület követelményekhez) jobban illeszkedő megoldás kerülhet implementálásra.

A HIG az I- és M-adatokat is teljesen analóg módon írja le, az egyetlen különbség az M-adatkapcsolatok esetén megjelenő tároló azonosító, amely lehetővé teszi, hogy több adatkapcsolat is ugyanazt a memóriahelyet használja.

Az I- és M-adatkapcsolatok közötti döntéssel kiegészített, funkcionális és imperatív bemeneteket is kezelni képes, általam javasolt rendszerszintű szintézis módszer (PipeComp) lépései az 1. ábrán láthatóak.

A dekompozíció során nem szükséges figyelembe venni, hogy a forráskódi leírásban I- vagy M-adatok szerepelnek. Amennyiben a dekompozíció két, adatfüggésben lévő csomópontot két külön partícióba oszt, ezek között biztosítani kell az adatok átvitelét a partíciók között. Az átvitelnél irreleváns, hogy I- vagy M-adatkapcsolatról van szó, mindkét esetben a teljes adatstruktúrát át kell mozgatni a partíciók között.

[3] `https://www.scala-lang.org`
[4] `https://kotlinlang.org`
[5] `https://www.eclipse.org/xtend`



1. ábra. Funkcionális és imperatív nyelvű rendszerszintű szintézis módszer

Miután a dekompozíciós fázis meghatározta, hogy a gráf mely csomópontjai mely feldolgozóegység számára kerüljön kiosztásra, az I- és M-adatkapcsolatok közötti döntés az egyes feldolgozóegységek tulajdonságait figyelembe véve elvégezhető. Szoftveres ágon érdemes kihasználni az M-adatok processzoros rendszerekben jelentkező előnyeit, mivel az I-adat használatából eredő párhuzamosítás ilyenkor nem játszik szerepet a szekvenciális futtatás miatt, ugyanakkor az M-adat szerinti megvalósításhoz kevesebb memória szükséges. Továbbá az I-adatokra jellemző másolások sem szükségesek ebben az esetben.

Hardveres ágon az I- és M-adatok közötti döntés a performanciát befolyásolja az előzőekben leírt okoknál fogva. A függőségi (D) és erőforrás (R) élek adatkapcsolatok alapján történő származtatása függ az I/M döntéstől, így ezen számításokat az I/M döntés utánra kell halasztani. Az ütemezés és allokáció a függőségi és erőforrás élek alapján történik.

2. tézis. *Kidolgoztam egy HIG alapú rendszerszintű szintézis módszert (PipeComp), amely a funkcionális és imperatív bemeneti nyelveket integráltan is képes kezelni. A dolgozat írásakor nem ismert olyan rendszerszintű- vagy magasszintű szintézis eszköz, amely mind a funkcionális, mind az imperatív paradigma szerinti nyelvek sajátosságait figyelembe veszi. A PipeComp algoritmusait modelltranszformációs lépések sorozatával írtam le. [1, 4, 5, 6, 7, 8]*

- 2.1: *Módszert adtam az I- és M-adatok egységes kezelésére a PipeComp szintézis során. A szakirodalomban szereplő tervezőrendszerekkel szemben nem a forráskód szintjén kell eldönteni, hogy a megvalósítás I- vagy M-adatkapcsolatokkal történjen, így a feladat megfogalmazásánál nem szükséges figyelembe venni a döntésnek az alsóbb absztrakciós szinteken jelentkező hatását. Ezáltal a felhasználó az algoritmustervezés során nagyobb szabadsági fokkal rendelkezik a legkedvezőbb leírásmód megválasztásában. Így az alsóbb szinten szükséges döntések későbbre halaszthatók az automatizálható optimumkeresést kedvezően befolyásolva. A megvalósítás során a megadott korlátokat (pl. időbeli és helykihasználás) figyelembe véve algoritmizált módon történhet az I- és M-adatkapcsolatok közötti döntés.*

- 2.2: *A PipeComp imperatív frontend esetében új, SSA (statikus, egyszeri hozzárendelési forma) alapú strukturált nyelvet definiáltam, amely a szoftveres C fordítók köztes reprezentációival ellentétben nem tartalmaz explicit vezérlésfolyamot. Ennek köszönhetően jobban illeszthető a HLS és SLS feladatokhoz, valamint a párhuzamosítható műveletek keresése és analízise is egyszerűbbé válik. A nyelv a szoftveres imperatív fordítórendszerek köztes reprezentációjaként is alkalmazható, egyszerűbbé téve az adatfolyam alapú optimalizálást.*

3.3. Egymásba ágyazott ciklusok és M-adatok pipeline ütemezése

A magasszintű- és rendszerszintű szintézis alkalmazása során, különösen digitális jelfeldolgozó (DSP, digital signal processing) rendszerek szintézise esetén az átbocsátóképesség rendkívül fontos tényező az elkészült rendszer hatékonyságának szempontjából. Az átbocsátóképesség növelésének egyik módszere a rendszer pipeline ütemezése [AVJ01].

A HLS és az SLS ütemezés fázisában [Chi12] dől el, hogy a gráf csomópontjai mikor kerülnek végrehajtásra. Az ütemezés történhet fordítási- vagy futási időben. Az előbbit statikus ütemezésnek, az utóbbit dinamikus ütemezésnek nevezzük. A

statikus ütemezés előnye, hogy már a fordítási időben meghatározásra kerül az egyes csomópontok végrehajtási ideje és az indítás kezdete, ezáltal egyszerűbb a hardver megvalósítása. A hátránya az, hogy amennyiben a műveletek végrehajtási ideje fordítási időben nem ismert vagy algoritmus útján nem kikövetkeztethető, maga az ütemezés sem végezhető el.

Statikus előütemezésnek olyan esetekben is van létjogosultsága, amikor a konkrét megvalósítás dinamikus ütemezéssel történik. Ilyenkor

- előzetes becslést lehet adni a műveletek végrehajtási és indulási idejére
- pipeline üzemmód esetén meghatározhatóak a művelet többszörözések és a szükséges újraindítási idők (iteration interval)
- az erőforrás és idő paraméterek közötti döntéseket már fordítási időben meg lehet hozni, becslést adva a várható eredményre

A pipeline ütemezés ciklusok esetén járhat hatékonyságnövelő eredménnyel, azaz ha a rendszert vagy annak egy részét egymás után többször, más-más adatokkal kell működtetni. Az ütemezés különösen bonyolulttá válhat, ha a rendszer egymásba ágyazott ciklusokból (nested loops) épül fel, vagy ha M-adatokkal kapcsolatos műveleteket is tartalmaz.

Egy rendszer maximális átbocsátóképességét úgy lehet növelni, hogy megkeressük azokat a komponenseket („szűk keresztmetszeteket”), amelyek az átbocsátóképességet (azaz az újraindítási időt) ténylegesen korlátozzák, majd ezeken változtatásokat végzünk. A hatékonyságnövelésre néhány lehetőség:

- ciklus esetén a ciklustörzs pipeline ütemezése
- a komponens többszörözése
- a komponens kicserélése hatékonyabb implementációra (ezzel a továbbiakban nem foglalkozunk, feltételezzük, hogy a komponens hardveres megvalósításához megfelelő implementációt választottunk)

A szűk keresztmetszetek megkeresése, és a használandó optimalizálási megoldással kapcsolatos döntések a mai HLS eszközökben általában a felhasználó feladata (ennek egyik módja a profiling). Az Irányítástechnika és Informatika tanszéken több, mint két évtized óta folyik kutatás a kívánt átbocsátóképesség (pontosabban újraindítási idő) elérése érdekében történő automatizált optimalizálás témakörében. Az eddigi kutatási eredmények viszont nem tértek ki az egymásba ágyazott ciklusok és az M- adatok algoritmizált pipeline optimalizálására.

A továbbiakban az egymásba ágyazott ciklusok és M-adatok statikus pipeline ütemezésével kapcsolatos kutatásaim eredményét ismertetem.

Az ütemezés legfőbb tulajdonságai:

3.1. Definíció. *Pipeline üzemmód: Egy HIG komponens pipeline üzemmódban működik, ha az előtt megkezdi egy következő bemeneti adat feldolgozását, mielőtt az előző adat feldolgozásának hatására minden kimeneti adat előállításra került.*

3.2. Definíció. *Lappangási idő: Egy HIG komponens lappangási ideje ($L(G)$) az az időtartam [6], amely a gráf bemeneteinek legkorábbi feldolgozása és a kimenetein megjelenő adatok legkésőbbi ideje között van (azaz több bemenet és több kimenet esetén a kimenetek idejének maximuma és a bemenetek idejének minimuma közötti idő).*

3.3. Definíció. *Újraindítási idő: Egy HIG újraindítási ideje ($R(G)$) az az időtartam, ami két, egymás utáni bemeneti adat feldolgozása között minimálisan eltelik.*

Egy adott HIG kiadódó újraindítási idejének meghatározásához a következő bemeneti információk szükségesek:

- minden művelet esetében a példányosított komponens végrehajtási ($t_D(n_i)$) és foglaltsági ($t_B(n_i)$) ideje
- minden erőforrás él foglaltsági ideje ($t_B(r_i)$)

$R(G)$ értékét a következő definíciókban három különböző esetre külön-külön definiálom. Elsőként a HIG pipeline üzemmód nélküli ütemezésére:

3.4. Definíció. *HIG minimális újraindítási ideje pipeline ütemezés nélkül (none-pipeline mode):*
$R_{NP}(G) = L(G)$

3.5. Definíció. *HIG minimális újraindítási ideje pipeline ütemezéssel, művelet többszörözés nélkül (pipeline mode, not replicated):*
$R_{PNR}(G) = \max\Big(\max_{\forall n_i \in N}\big(t_B(n_i)\big), \max_{\forall r_i \in E_R}\big(t_B(r_i)\big)\Big)$

3.6. Definíció. *HIG minimális újraindítási ideje pipeline ütemezéssel, művelet és erőforrás többszörözéssel:*

$$
\begin{aligned}
&R_{MIN}(G) = \max\Big(\max_{\forall n_i \in N}\big(t_{MINBN}(n_i)\big), \max_{\forall r_i \in E_R}\big(t_{MINBE}(r_i)\big)\Big)\\
&t_{MINBN}(n_i) := \begin{cases} 1 & \textit{ha } n_i \textit{ többszörözhető}\\ t_B(n_i) & \textit{ha } n_i \textit{ nem többszörözhető}\end{cases}\\
&t_{MINBE}(r_i) := \begin{cases} 1 & \textit{ha } r_i \textit{ többszörözhető}\\ t_B(r_i) & \textit{ha } r_i \textit{ nem többszörözhető}\end{cases}
\end{aligned}
\tag{1}
$$

[6] az időtartamot órajel ciklusokban szokás mérni

A HIG minimális újraindítási idejének meghatározásához a példányosított komponensek t_D és t_B paraméterei szükségesek (megj.: t_D az $L(G)$ kiszámítása miatt szükséges), így a következőkben minden komponenstípus esetére megadom ezek kiszámítási módját.

Az elemi komponensek végrehajtási (t_D) és foglaltsági (t_B) idejét paraméterként adottnak tekintjük, így ezzel a továbbiakban nem foglalkozunk. HIG komponens esetén a végrehajtási idő a belső gráf lappangási idejével, míg foglaltsági ideje a belső gráf újraindítási idejével egyezik meg:

$$t_D(G) = L(G) \tag{2}$$
$$t_B(G) = R(G) \tag{3}$$

Ez azt jelenti, hogy egymásba ágyazott HIG-ek esetén egy adott HIG pipeline újraindítási idejének meghatározása előtt minden tartalmazott, belső HIG komponens foglaltsági idejét, végeredményben újraindítási idejét is meg kell határozni. A HIG komponensek fa hierarchiájában először a levelekkel kell kezdeni a számítást, azaz egy lentről-fel (bottom-up) rekurzív algoritmusról van szó.

Elágazás komponens esetén a végrehajtási és foglaltsági idő az egyes belső HIG-ek paramétereinek maximumával írható le:

$$\begin{aligned} t_D &= \max(t_D(C_i)) \\ t_B &= \max(t_B(C_i)) \end{aligned} \tag{4}$$

Ciklus komponens esetén figyelembe kell venni a ciklus iterációs számát (tc), a ciklus törzsének foglaltsági (t_B) és végrehajtási (t_D) idejét:

$$t_B = t_D = (tc - 1){t_B}'(B) + t_D(B) \tag{5}$$

Az 5. képletben a ciklus törzsének foglaltsági ideje ($t_B(B)$) helyett ${t_B}'(B)$ szerepel, ebben az esetben ugyanis nem elég a HIG komponens foglaltsági idejével számolni, a ciklus visszacsatolások (LCD, loop carried dependencies [STL04]) is korlátozzák a minimális pipeline újraindítási időt. A visszacsatolásokkal kiterjesztett foglaltsági idő a komponens foglaltsági idejének és a visszacsatolások időtartamainak maximuma:

$${t_B}'(B) = \max\left(t_B(B), \max_{\forall i \in [1,n]}\left(t_{PT}(y_i) - t_{CT}(f_i)\right)\right) \tag{6}$$

A képlet szerint a visszacsatolás időtartamát a visszacsatoláshoz tartozó kimeneti és bemeneti portok időzítési különbségével adtuk meg. Emiatt szükséges, hogy a

transzformáció feltételében megadott t_{CT} és t_{PT} port attribútumok is rendelkezésre álljanak.

Az 5. képlet a következőképp magyarázható: míg a ciklus komponens egyszer hajtódik végre, a belső gráf tc-szer, tehát az első és utolsó újraindítás között $(tc - 1) * t_B{}'(B)$ idő telik el. Az utolsó újraindítás után a ciklus komponens még foglalt marad $t_D(B)$ ideig (ennyi idő kell az utolsó bemenet teljes feldolgozásához), így a két érték összege adja a loop komponens végrehajtási idejét. A foglaltsági időt ciklus esetében a kiadódó végrehajtási idő adja meg.

Egy rendszerrel szemben támasztható fontos követelmény lehet egy megadott áteresztőképesség teljesítése, másképpen megközelítve az egymást követő bemeneti adatok feldolgozása közti időintervallum adott értékének biztosítása. Ez utóbbi az elvárt újraindítási idő (R_D). A követelmény teljesítése a rendszer módosítását, és adott esetben a felhasznált hardveres erőforrások számának növekedését vonhatja maga után. Egyszerű (nem hierarchikus) gráfok esetén puffer behelyezéssel és művelet többszörözéssel elérhető a kívánt áteresztőképesség [AVJ01]. A következőkben a HIG specialitásaira (M-adatkapcsolatok, hierarchikus gráfok, Loop és Sel komponensek) is definiálom az adott pipeline újraindítási időre történő rendszertervezést.

Elemi komponensek esetén a foglaltsági idő (t_B) csökkentésének módja a művelet többszörözése. HIG komponensek foglaltsági idejének csökkentését a HIG komponens pipeline ütemezésével lehet elérni. Ha R_D kisebb, mint a kiszámított R_{NP}, akkor minden olyan komponenst és erőforrást, amely R_D elérését korlátozza (azaz, amire $t_D > R_D$), módosítani kell a komponens típusa szerint. Elágazás komponens esetén a kívánt újraindítási időt értelemszerűen minden belső komponensre külön kell értelmezni.

Ciklus komponens esetén a teljes komponens többszörözésénél létezik hatékonyabb megoldás, a ciklus pipeline ütemezése. Ezzel a módszerrel a ciklus törzsére (B) állapítható meg korlát a ciklusra (C) vonatkoztatott maximális foglaltsági idő alapján:

$$R_D(B) = \max\left(\left\lfloor \frac{R_D(C) - t_D(B)}{tc(C) - 1} \right\rfloor, R_{MIN}(B)\right) \tag{7}$$

Amennyiben a visszacsatolások is korlátozzák a foglaltsági időt, a visszacsatoláshoz tartozó bemeneti- és kimeneti portok közötti kritikus úton a műveletek végrehajtási idejét szükséges csökkenteni.

Annak érdekében, hogy a legkisebb költségű megoldást kapjuk, a kritikus úton lévő összes csomópontra ki kell számolni az egy egységnyi végrehajtási idő csökkentéséhez szükséges plusz költséget. Ezek után azt kell választani, amely az egységnyi

végrehajtási idő csökkentéséhez a legkisebb költséggel rendelkezik. Ezzel a módszerrel addig kell csökkenteni a végrehajtási időt, amíg a kritikus úton el nem érjük a kívánt végrehajtási időt. (Megj: az egy egységnyi végrehajtási idő csökkentése valójában nagyobb végrehajtási idő változást eredményezhet, hasonlóan az újraindítási idő 1-1 egységgel történő csökkentéséhez.)

Az komponensek végrehajtási idejének csökkentése az alábbi módon történhet, a komponens típusok szerinti esetekre vonatkozóan:

- Elemi komponens: a végrehajtási idő csökkentésének egyedüli módja új implementáció választása, így ezzel az esettel a továbbiakban nem foglalkozunk.
- HIG komponens: a kritikus úton lévő csomópontok végrehajtási idejét kell csökkenteni, az imént leírt megoldással analóg módon.
- Elágazás komponens: a belső HIG komponensek végrehajtási idejét kell csökkenteni, az eredő költség az egyes HIG komponensekre vonatkozott költségek összege.
- Ciklus komponens: pipeline ütemezéssel csökkenthető a végrehajtási idő, lásd a jelen alfejezet ciklusokról szóló szakaszát.

Ha az elvárt újraindítási időt az M-adatok miatt létrehozott erőforrásél is korlátozza, akkor két lehetőség adódik: az erőforrás él forrás- és célműveletei között a kritikus út végrehajtási idejét kell csökkenteni az előző fejezetben bemutatott módon, vagy erőforrás többszörözést és erőforrás kijelölő adatkapcsolatokat kell létrehozni.

3. tézis. *Módszert adtam a HIG köztes nyelven leírt műveletek statikus, hierarchikus pipeline ütemezésére, amely a kívánt újraindítási idő (iterációs intervallum) elérése érdekében a gráfhierarchia szükséges szintjein végez módosításokat. A módszer kidolgozásával lehetővé vált az I- és M-adatokat, valamint az adattól nem függő (vagy adatfüggő, de korlátos) iterációszámú egymásba ágyazott ciklusokat is tartalmazó feladatok hierarchikus pipeline ütemezése. A módszer első lépése során minden HIG komponens esetében a pipeline üzem nélküli újraindítási idő kerül kiszámításra. Egy adott HIG komponens akkor kerül pipeline ütemezésre, ha a teljes rendszer elvárt újraindítási idejének teljesítése érdekében erre szükség van. Amennyiben a pipeline ütemezés sem elégséges a követelmények teljesítéséhez, a művelet többszörözésére kerül sor. Ciklus-visszacsatolások korlátozása esetén a kritikus úton lévő belső ciklusok pipeline ütemezése adhat megoldást. [2, 8]*

4. Új eredmények alkalmazása

A magasszintű szintézis területen a legismertebb és legtöbbet használt benchmark készlet a CHStone [Har08]. A készlet tizenkét, C nyelven írt programkódot tartalmaz, amelyek mindegyike ismert algoritmust implementál a kommunikációs protokollok, titkosítás, média fájlok és aritmetikai részterületekről. A CHStone készletből az SHA (Secure hash algorithms) algoritmus példáján keresztül mutatom be a tézisek gyakorlati hasznosíthatóságát, ez ugyanis minden, a pipeline ütemezési módszer bemutatásához szükséges komponenst tartalmaz, ugyanakkor megfelelően átlátható.

A tézisekben bemutatott módszerekre épülve megterveztem és implementáltam a PipeComp HLS rendszert. Ehhez a munkához a következő technológiákat használtam fel:

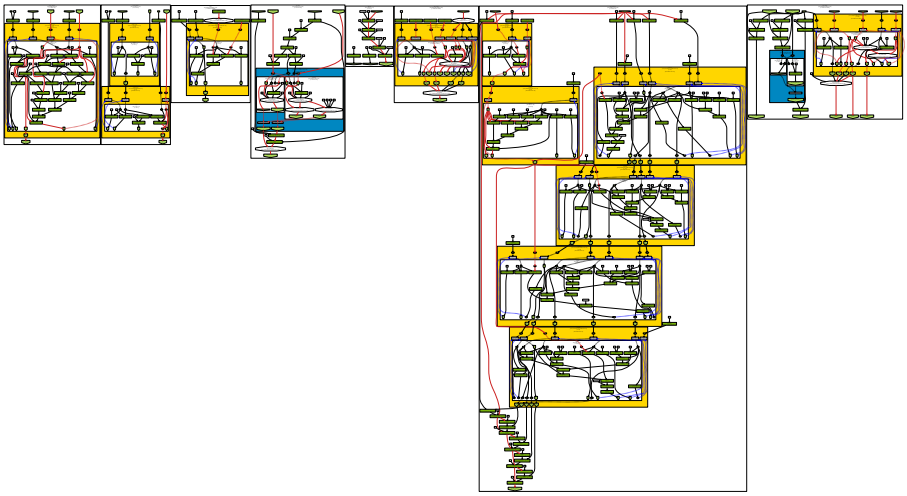
- Java 8: a kód nagyrésze Java nyelven készült, az 1.8-as verzióra épülve
- ANTLR[7]: parszer generátor, C forráskódok szintaktikai elemzéséhez
- EMF (Eclipse Modeling Framework)[8] [SBPM09]: modell alapú fejlesztésre szolgáló keretrendszer
- VIATRA[9] [Újh16]: gráf alapú modellek validálásához és transzformációjához
- Xtend[10]: modell manipuláláshoz és kódgeneráláshoz
- Xtext[11]: szöveges DSL feldolgozásához, kezeléséhez
- GraphViz[12]: gráfok grafikus rendereléséhez (az értekezésben bemutatott HIG grafikus diagramok ezzel az eszközzel készültek)

Az SHA benchmarkból a modelltranszformációk során a 2. ábra felső részén látható HIG készült el. A pipeline ütemezés összefoglaló táblázata és költségfüggvény az ábra alsó felében látható.

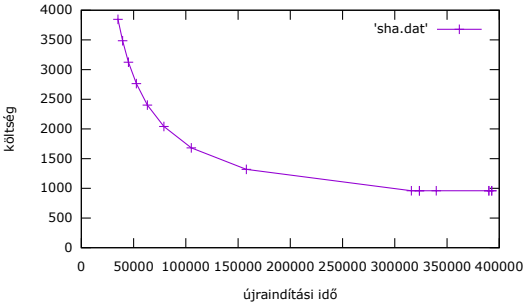
4.1. Ipari alkalmazás

A 2. tézisben összefoglalt PipeComp fordítórendszer C nyelvű frontendje, ill. az ehhez használt SSA alapú imperatív modell a Prolan Zrt.-ben a gyakorlatban is hasznosításra került. A cég többek között biztonságkritikus vasúti irányítórendszereket fejleszt, amely során a beágyazott szoftverek C nyelvű forráskódjainak feldolgozására és elemzésére volt szükség memórialeírások készítése céljából.

[7] `https://www.antlr.org/`
[8] `https://www.eclipse.org/emf`
[9] `https://www.eclipse.org/viatra`
[10] `https://www.eclipse.org/xtend`
[11] `https://www.eclipse.org/Xtext`
[12] `https://www.graphviz.org`



R	C	main	shastream	hig62	hig21	hig141	hig134	loop63node3
392842	959	-	-	-	-	-	-	-
392822	959	P	-	-	-	-	-	-
390032	959	P	P	-	-	-	-	-
390030	959	P	P	P	-	-	-	-
339738	959	P	P	-	P	-	-	-
323612	959	P	P	-	P	P	-	-
323610	959	P	P	P	P	P	-	-
315930	959	P	P	-	P	P	P	-
157965	1320	P	P	P	P	P	P	2
105310	1681	P	P	P	P	P	P	3
78983	2042	P	P	P	P	P	P	4
63186	2403	P	P	P	P	P	P	5



2. ábra. SHA pipeline ütemezés összefoglaló táblázat és költségfüggvény

5. Publikációk

Saját publikációk

[1] Péter Arató and Gergely Suba. Data Flow Graph Generation of Nested Loops from Imperative High Level Description. *SCIENTIFIC BULLETIN of The POLITEHNICA University of Timişoara, Romania*, 59(73)(2):123–130, 2014.

[2] Gergely Suba. Hierarchical Pipelining of Nested Loops in High-Level Synthesis. *Periodica Polytechnica Electrical Engineering and Computer Science*, 58(3):81–91, 2014.

[3] Gergely Suba and Péter Arató. A new data flow graph model extended for handling loops and mutable data in high level synthesis. *Workshop on the Advances of Information Technology*, pages 1–5, 2017.

[4] Péter Arató and Gergely Suba. A data flow graph generation method starting from C description by handling loop nest hierarchy. In *IEEE 9th IEEE International Symposium on Applied Computational Intelligence and Informatics (SACI)*, pages 269–274. IEEE, may 2014.

[5] Gergely Suba and Péter Arató. A new method for transforming algorithm into VHDL by starting from a Haskell functional language description. In *Middle-European Conference on Applied Theoretical Computer Science*, pages 10–11, 2016.

[6] Gergely Suba and Péter Arató. Concept of the system-level synthesis framework PipeComp. In *WAIT 2015 : Workshop on the Advances of Information Technology*, Budapest, 2015.

[7] Péter Arató, István Loványi, Gergely Patai, Gergely Suba, Timotity Milan, and Tamás Torba. Compiling High Level Flow Languages into Hardware. Technical report, 2010.

[8] Péter Arató, Péter Dóbé, Daniel Andras Drexler, András Ercsényi, Balázs Goldschmidt, Tamás Horváth, Richárd Péter Kápolnai, István Loványi, György Pilászy, György Rácz, Balázs Simon, Ágoston Srp, and Gergely Suba. Feladatfüggő felépítésű pipeline többprocesszoros rendszerek tervezési módszerének kidolgozása és alkalmazása nagy sebességigényű beágyazott célrendszerekben - TÁMOP-4.2.2.C-11/1/KONV-2012-0004 projekt dokumentáció. Technical report, Pannon Egyetem, 2015.

Hivatkozások

[AANS+14] Oriol Arcas-Abella, Geoffrey Ndu, Nehir Sonmez, Mohsen Ghasempour, Adria Armejach, Javier Navaridas, Wei Song, John Mawer, Adrian Cristal, and Mikel Lujan. An empirical evaluation of High-Level Synthesis languages and tools for database acceleration. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8. IEEE, sep 2014.

[AVJ01] Péter Arató, Tamás Visegrády, and István Jankovits. *High Level Synthesis of Pipelined Datapaths*. John Wiley & Sons, Ltd, New York, USA, 2001.

[Awa09] Mariette Awad. FPGA supercomputing platforms: A survey. In *2009 International Conference on Field Programmable Logic and Applications*, pages 564–568. IEEE, aug 2009.

[BCSS98] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware Design in Haskell. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming - ICFP '98*, volume 34, pages 174–184, New York, New York, USA, jan 1998. ACM Press.

[BEH+] Ulrik Brandes, Markus Eiglsperger, Ivan Herman, Michael Himsolt, and M. Scott Marshall. GraphML Progress Report.

[BKK+10] Christiaan Baaij, Matthijs Kooijman, Jan Kuper, Arjan Boeijink, and Marco Gerards. CLaSH: Structural Descriptions of Synchronous Hardware Using Haskell. In *2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*, pages 714–721. IEEE, sep 2010.

[BP03] Gianfranco Bilardi and Keshav Pingali. Algorithms for computing the static single assignment form. *Journal of the ACM*, 50(3):375–425, may 2003.

[But14] Paul Butcher. *Seven Concurrency Models in Seven Weeks.* Cambridge University Press, Cambridge, 2014.

[Cam90] R. Camposano. From behavior to structure: High-level synthesis. *Design & Test of Computers, IEEE*, 7(5):8–19, oct 1990.

[CCA+11] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. LegUp: High-level synthesis for FPGA-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays - FPGA '11*, page 33, New York, New York, USA, 2011. ACM Press.

[CCB+08] Philippe Coussy, Cyrille Chavet, Pierre Bomel, Dominique Heller, Eric Senn, and Eric Martin. GAUT: A High-Level Synthesis Tool for DSP Applications. In Philippe Coussy and Adam Morawiec, editors, *High-Level Synthesis*, pages 147–169. Springer Netherlands, 2008.

[CDL11] Alexandre Cornu, Steven Derrien, and Dominique Lavenier. HLS tools for FPGA: Faster development with better performance. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 6578 LNCS, pages 67–78, 2011.

[CDW10] João M. P. Cardoso, Pedro C. Diniz, and Markus Weinhardt. Compiling for reconfigurable computing. *ACM Computing Surveys*, 42(4):1–65, jun 2010.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, oct 1991.

[CGMT09] Philippe Coussy, Daniel D. Gajski, Michael Meredith, and Andres Takach. An Introduction to High-Level Synthesis. *IEEE Design & Test of Computers*, 26(4):8–17, jul 2009.

[Chi12] M Chinnadurai. Survey On Scheduling And Allocation In High Level Synthesis. *International Journal of Computer Science & Engineering Survey*, 3(5):43–51, oct 2012.

[CNNV11] Jason Cong, Stephen Neuendorffer, Juanjo Noguera, and Kees Vissers. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, apr 2011.

[Edw05] Stephen A. Edwards. The challenges of hardware synthesis from C-like languages. In *Proceedings -Design, Automation and Test in Europe, DATE '05*, volume I, pages 66–67. Ieee, 2005.

[EGH+05] D Edelsohn, W Gellerich, M Hagog, D Naishlos, M Namolaru, E Pasch, H Penner, U Weigand, and A Zaks. Contributions to the GNU Compiler Collection. *IBM Systems Journal*, 44(2):259–278, 2005.

[GAGS09] Daniel D. Gajski, Samar Abdi, Andreas Gerstlauer, and Gunar Schirner. *Embedded System Design*, volume 35. Springer US, Boston, MA, 2009.

[GHP+09] Andreas Gerstlauer, Christian Haubelt, A.D. Pimentel, T.P. Stefanov, Daniel D. Gajski, and J. Teich. Electronic System-Level Synthesis Methodologies. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(10):1517–1530, oct 2009.

[Har08] Yuko Hara. CHStone: A benchmark program suite for practical C-based high-level synthesis. In *2008 IEEE International Symposium on Circuits and Systems*, pages 1192–1195. IEEE, may 2008.

[HDA08] Scott Hauck, André Dehon, and DeHon André. *Reconfigurable Computing - The Theory and pratice of FPGA-BASED computing.* 2008.

[JHM04] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Computing Surveys*, 36(1):1–34, mar 2004.

[KM13] Nikolaos Kavvadias and Kostas Masselos. The HercuLeS high-level synthesis environment. In *2013 23rd International Conference on Field programmable Logic and Applications*, pages 1–1. IEEE, sep 2013.

[LA03] Chris Lattner and Vikram Adve. Architecture for a Next-Generation GCC. *GCC Developers Summit*, page 121, 2003.

[LA04] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, number c, pages 75–86. IEEE, 2004.

[Mer03] Jason Merrill. Generic and gimple: A new tree representation for entire functions. *Proceedings of the 2003 GCC Developers' Summit*, 2003.

[MPC90] M.C. McFarland, A.C. Parker, and R. Camposano. The high-level synthesis of digital systems. *Proceedings of the IEEE*, 78(2):301–318, 1990.

[MST+09] Grant Martin, Gary Smith, Don Tho, Mario Barbacci, and Alice Parker. High-Level Synthesis: Past, Present, and Future. *IEEE Design & Test of Computers*, 26(4):18–25, jul 2009.

[MVG+12] Wim Meeus, Kristof Van Beeck, Toon Goedemé, Jan Meel, and Dirk Stroobandt. An overview of today's high-level synthesis tools. *Design Automation for Embedded Systems*, 16(3):31–51, aug 2012.

[NLG99] Walid a. Najjar, Edward a. Lee, and Guang R. Gao. Advances in the dataflow computational model. *Parallel Computing*, 25(13-14):1907–1929, dec 1999.

[NRE04] R. Namballa, N. Ranganathan, and A. Ejnioui. Control and data flow graph extraction for high-level synthesis. In *IEEE Computer Society Annual Symposium on VLSI*, pages 187–192. IEEE Comput. Soc, 2004.

[NSP+16] Razvan Nane, Vlad-mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Brown, Fabrizio Ferrandi, Jason Anderson, and Koen Bertels. A Survey and Evaluation of FPGA High-Level Synthesis Tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(10):;article–PhD1591–1604, oct 2016.

[OMG15] OMG. OMG Unified Modeling Language (Version 2.5). Technical report, 2015.

[PF13] Christian Pilato and Fabrizio Ferrandi. Bambu: A modular framework for the high level synthesis of memory-intensive applications. In *2013 23rd International Conference on Field programmable Logic and Applications*, pages 1–4. IEEE, sep 2013.

[SBPM09] Dave Steinberg, Frank Budinsky, Marcelo Paternoostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2009.

[SK86] J Sargeant and CC C Kirkham. Stored data structures on the Manchester dataflow machine. *ACM SIGARCH Computer Architecture News*, 14(2):235–242, jun 1986.

[SL16] Hayden Kwok Hay So and Cheng Liu. *FPGAs for Software Programmers*. Springer International Publishing, Cham, 2016.

[STL04] H. Styles, D.B. Thomas, and W. Luk. Pipelining designs with loop-carried dependencies. In *Proceedings. 2004 IEEE International Conference on Field- Programmable Technology (IEEE Cat. No.04EX921)*, pages 255–262. IEEE, 2004.

[TC98] Giri Tiruvuri and Moon Chung. Estimation of lower bounds in scheduling algorithms for high-level synthesis. *ACM Transactions on Design Automation of Electronic Systems*, 3(2):162–180, apr 1998.

[TC11] Andrew Tolmach and Tim Chevalier. An External Representation for the GHC Core Language. Technical report, The GHC Team, 2011.

[TCW+05] T.J. Todman, G.A. Constantinides, S.J.E. Wilton, O. Mencer, W. Luk, and P.Y.K. Cheung. Reconfigurable computing: architectures and design methods, 2005.

[TGP07] Justin L. Tripp, Maya B. Gokhale, and Kristopher D. Peterson. Trident: From High-Level Language to Hardware Circuitry. *Computer*, 40(3):28–37, mar 2007.

[Újh16] Zoltán Újhelyi. *Program Analysis Techniques for Model Queries and Transformations*. PhD thesis, 2016.

[VN14] Mario Vestias and Horacio Neto. Trends of CPU, GPU and FPGA for high-performance computing. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–6. IEEE, sep 2014.

[WKR02] Andreas Winter, Bernt Kullbach, and Volker Riediger. Software Visualization. In Stephan Diehl, editor, *Software Visualization*, volume 2269 of *Lecture Notes in Computer Science*, pages 324–336. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

[WP94] P.G. Whiting and R.S.V. Pascoe. A history of data-flow languages. *IEEE Annals of the History of Computing*, 16(4):38–59, jan 1994.