Višekriterijska pretraga prostora oblikovanja heterogenih višeprocesorskih platforma zasnovana na elementarnim operacijama

Frid, Nikolina

Sveučilište u Zagrebu

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Nikolina Frid

Višekriterijska pretraga prostora oblikovanja heterogenih višeprocesorskih platforma zasnovana na elementarnim operacijama

DOKTORSKI RAD

Zagreb, 2019.

Sveučilište u Zagrebu

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Nikolina Frid

Višekriterijska pretraga prostora oblikovanja heterogenih višeprocesorskih platforma zasnovana na elementarnim operacijama

DOKTORSKI RAD

Mentor: izv. prof. dr. sc. Vlado Sruk

Zagreb, 2019.

University of Zagreb

FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

Nikolina Frid

Multiobjective design space exploration of heterogeneous multiprocessor platforms based on elementary operations

DOCTORAL THESIS

Supervisor: Associate professor Vlado Sruk, PhD

Zagreb, 2019

Doktorski rad izrađen je na Sveučilištu u Zagrebu Fakultetu elektrotehnike i računarstva, na Zavodu za elektroniku, mikroelektroniku, računalne i inteligentne sustave

Mentor: izv. prof. dr. sc. Vlado Sruk

Doktorski rad ima: 159 stranica

Doktorski rad br.: _____

O mentoru

Vlado Sruk rođen je u Zagrebu 1965. godine. Diplomirao je, magistrirao i doktorirao u polju elektrotehnike na Sveučilištu u Zagrebu Fakultetu elektrotehnike i računarstva (FER), 1988., 1991. odnosno 1998. godine.

Od siječnja 1989. godine radi na Zavodu za elektroniku, mikroelektroniku, računalne i inteligentne sustave FER-a. U svibnju 2007. godine izabran je u zvanje izvanrednog profesora. Znanstveni i stručni interesi uključuju područja višeprocesorskih ugradbenih sustava, mobilno računarstvo te računarstvo visokih performansi s naglaskom na arhitekture memorija, pouzdanost i razvoj programske potpore za ugradbene sustave.

Sudjelovao je na pet znanstvenih projekata Ministarstva znanosti, obrazovanja i sporta Republike Hrvatske, dva EU FP7 projekta i jednom H2020 projektu. Trenutno sudjeluje na međunarodnom projektu "The European Processor Initiative (EPI)" pod pokroviteljstvom Europske Komisije. Objavio je više od 30 radova u časopisima i zbornicima konferencija u području oblikovanja ugrađenih sustava i razvoja programske potpore.

Član je strukovnih udruga IEEE i ACM. Sudjeluje u radu međunarodnih programskih odbora znanstvenih konferencija, te sudjeluje kao recenzent u većem broju inozemnih časopisa i međunarodnih konferencija.

About the Supervisor

Vlado Sruk was born in Zagreb in 1965. He received B.Sc., M.Sc. and Ph.D. degrees in electrical engineering from the University of Zagreb, Faculty of Electrical Engineering and Computing (FER), Zagreb, Croatia, in 1988, 1991 and 1998, respectively.

From January 1989, he is working at the Department of Electronics, Microelectronics, Computer and Intelligent Systems at FER. In May 2007 he was promoted to Associate Professor. His research focuses mainly on the areas of multi-core embedded systems, mobile computing, and high performance computing with emphasis on memory architectures, state-of-the-art concepts and techniques in multicore software engineering and fault-tolerant computing.

He participated in six scientific projects financed by the Ministry of Science, Education and Sports of the Republic of Croatia, two EU FP7 projects and one H2020 project. Currently he is involved in research project "The European Processor Initiative (EPI)" under sponsorship from European Comission. He published more than 30 papers in journals and conference proceedings in the area of embedded system design and software design.

He is a member of IEEE and ACM. He participated in work of conference international programs committees and he serves as a technical reviewer for various international journals and conferences.

Sažetak

Doktorski rad bavi se problemom pretrage prostora oblikovanja heterogenih višeprocesorskih platformi kao ključnim dijelom procesa razvoja. U radu se predlaže metoda pretrage prostora oblikovanja koja obuhvaća postupak rane procjene trajanja izvođenja te heuristika za rješavanje optimizacijskog problema raspoređivanja u heterogenim MPSoC sustavima. Rana procjena trajanja izvođenja aplikacije temelji se na konceptu *elementarnih operacija* koji omogućava određivanje trajanja izvođenja pojedinih operacija na različitim platformskim konfiguracijama bez izrade modela procesorskog podatkovnog puta i priručne memorije. Korištenjem elementarnih operacija izgrađuju se apstraktni modeli aplikacije i višeprocesorske platforme. Ti modeli se koriste u heurističkoj metodi pretrage prostora oblikovanja koja se temelji na evolucijskom algoritmu NSGA-II uz prilagodbu specifičnostima heterogenih MPSoC sustava. Pri raspoređivanju dijelova aplikacije na elemente platforme, optimiraju se izračun i komunikacija istovremeno prema dva kriterija: vremenu izvođenja i zauzeću. Predloženu metodu karakteriziraju modularnost, skalabilnost i ponovna uporabivost čime se postiže smanjenje jaza između brzine dobivanja potencijalnih rješenja i točnosti procjene njihovih performansi.

Ključne riječi: pretraga prostora oblikovanja, heterogene platforme, MPSoC, procjena trajanja izvođenja, elementarne operacije, evolucijski algoritmi, NSGA-II

Abstract

Multiobjective design space exploration of heterogeneous multiprocessor platforms based on elementary operations

The doctoral dissertation is the result of the research in the field of design space exploration as the key part in the development of heterogeneous multiprocessors systems. Due to the ever increasing complexity of embedded systems applications, which pervade the modern world, it is required that each new generation provides more and more advanced functionality. Designing heterogeneous multiprocessor systems is a major challenge due to the number and variety of components, which significantly increase the size of the design space, i.e. the number of possible solutions. This creates a gap between the speed of obtaining potential solutions and the accuracy of the evaluation of their performance. In order to reduce the gap, it is necessary to model the entire system using high-level abstraction models which can be then used to obtain a preliminary performance assessment of possible solutions at a very early stage and prone the design space quickly.

In this dissertation a new design space exploration method is proposed, specially targeting heterogeneous multiprocessor systems. It includes an early estimation of runtime performance and heuristics to solve an optimization mapping and scheduling optimization problem for heterogeneous MPSoC systems.

Early estimation of application execution time is based on the novel concept of elementary operations, which enables estimation of the duration of execution of individual operations on different platform configurations without creating a processor data path and cache models. Using elementary operations, abstract models of application and multiprocessor platforms are built. These models are used in a heuristic design space exploration method based on the NSGA-II evolutionary algorithm with adaptation to the specifics of heterogeneous MPSoC systems. When mapping parts of an application to platform elements, computation and communication are optimized simultaneously according to two criteria: runtime and occupancy. Furthermore, a special case of platforms - the so-called "sparsely connected platforms", where processing elements are not connected to every memory element, is also considered and included in the

proposed design space exploration method.

Overall, the proposed method enables a modular and scalable approach to design space exploration which is based on high abstraction level and high reusability of intermediate results, thereby reducing the gap between the speed of obtaining potential solutions and the accuracy of their performance evaluation.

Chapter 1 introduces the basic concepts like embedded platforms, heterogeneity etc. and presents the motivation behind the research conducted. The contributions of the doctoral research and an overview of the work are also outlined.

Chapter 2 presents the basic theory of modern approaches to embedded systems design and development, with focus on abstraction levels and existing design methodologies. Three major embedded system development methodologies are presented, with a special emphasis on System-level design methodology (SLD), which is particularly suited for the development of heterogeneous multiprocessor systems. At the core of all these methodologies is the possibility of early performance estimation and model evaluation to enable early decision making and quick design space pruning.

Chapter 3 addresses the issue of estimating application execution duration solely based on the source code. First, the state of art in the field of source-level timing estimation is presented. Then, the proposed method for estimating the duration of the application execution - called ELementary OPerationS based Estimation Method (ELOPS-EM) is presented.

Central to the proposed method is introduction of the novel concept - elementary operations, defined as syntax units that are viewed as a whole in the source code of applications written in C, and whose execution time on the platform can be measured completely independently of other parts of the code and the application itself in which they appear. Different code statements are classified as various types of elementary operations, and in this way, profiles of all applications and platforms are created. Based on these models, the duration of the entire application can be estimated without the need for simulation at the basic block level and without the development of a mathematical model of the processor data path and cache, which is required by most current approaches. Application and platform profiles are completely independent of one another and it is therefore possible to reuse previously acquired profiles for different combinations of applications and platforms.

The method has been evaluated in an environment that represents a typical real-world heterogeneous multiprocessor system. The JPEG compression algorithm and the AES encryption

algorithm were selected as test applications, and the target test platform was Xilinx Zynq ZC706 in several different configurations. Overall, timing estimation accuracy is nearly at the same level for all test cases: average error is around 5%, and maximum error remains below 17%, which is comparable to other state of art source-level timing estimation methods with having additional profile reusability to minimize future time and effort. The amount of underestimated timing values is around the same as the amount of overestimated values, leaving it to future research to investigate further in which particular cases does the ELOPS-EM method give under or over estimated values. From the aspect of compiler code optimization, there seems to be no difference in the estimation accuracy for different optimization levels.

The topic of Chapter 4 is design space exploration of heterogeneous embedded systems. At the beginning, an overview of the state of art is given. This is followed by a presentation of the proposed system model and the proposed design space exploration method, as well as the evaluation of the results on artificial and real test cases.

The proposed method for design space exploration is based on a high-level system model, following the principles of system-level design (SLD), which assumes having separate application and platform models and separating computation from communication in those models. The system model includes application and platform models which are built based on the concept of elementary operations and the application and platform profiles described in the previous chapter.

These models are used in a heuristic design space exploration method that aims to find the optimal scheme for mapping parts of an application to platform elements and to determine the schedule of their execution. Considering the great impact of memory configuration on the performance of the entire system, both computation and communication are mapped: computation to processing elements and communication channels to memory elements. The method is based on the evolutionary algorithm for multi-criteria optimization - NSGA-II, and two criteria are optimized: runtime and total number of platform elements used. Two variants of design space exploration method are considered: simultaneous mapping (SDSE) and two-stage mapping (2SDSE). In the simultaneous mapping approach - SDSE, procedures (i.e. computation) are mapped to processors and communication channels between procedures are mapped to memory at the same time. Alternatively, the two-stage mapping - 2SDSE is performed in two iterations: first, the procedures are mapped to the processors and then the communication channels are mapped to the memory elements.

The evaluation has been done on both artificially created models and models of real applications and platforms. The selected real applications have been JPEG compression and Ray Tracing, and the platforms have been Xilinx ZC706 and Adapteva Parallella. Three evaluation measures are used: hypervolume, inverted generational distance and correlation with reference front. Overall, SDSE algorithm is undoubtedly more successful than the 2SDSE algorithm according to all three measures. In certain cases, where the communication to computation ratio is very low and there are many procedures which cannot be executed on all processors, both algorithms give equally good results which is expected given the fact that communication plays a much less significant role and the number of feasible solutions in relatively low.

At the end of the chapter, a special consideration is given to the case when not all processors are connected to all memory elements, i.e. the platform is sparsely connected. In such case, the number of infeasible solutions can significantly exceed the number of feasible solutions, which makes the design space exploration even more complex. In such cases, it has been shown that the SDSE algorithm cannot always find feasible solutions, and thus the SDSE algorithm has been varied into four additional versions specifically tailored to such platforms. The performance of each version has been evaluated on specially crafted artificial models and the results show that the initial version of SDSE algorithm is overall the most successful in finding solutions, especially in cases when not all procedures can be executed on all processors. But, other two versions of the algorithm: C-SDSE and MCN-SDSE, which are designed to target clusters of processors on such platforms, outperform SDSE when the problem is slightly relaxed so that most procedures can execute on all processors. Also, there is a certain level of complementarity between SDSE and MCN-SDSE - in cases where SDSE cannot find any feasible solutions, MCN-SDSE has a great chance to find a solution and vice versa. This demonstrates that SDSE algorithm is most versatile, but introducing certain modifications tailored to suit specific platform configurations, like targeting clusters of processors, can yield better results in such situations.

Chapter 5 concludes the dissertation and provides final considerations. In it, the realization of expected contributions of the dissertation is briefly analysed, and a guidance for future research is provided, which will be mainly in the direction of extending the method for source level timing estimation on other platform architectures like DSP and CISC, and further improvement of the design space exploration method to suit better the different types of sparsely connected platforms.

To summarize, the three major contributions of this doctoral thesis are:

1. A new method for application source code classification and platform profiling based on the concept of elementary operations.
2. High level application and hardware platform models built from elementary operations profiles and used for analytical estimation of application execution duration.
3. A new multiobjective design space exploration method for heterogenenous systems which employs the high level application and platform models.

Keywords: design space exploration, heterogeneous platforms, MPSoC, timing estimation, elementary operations, evolutionary algorithms, NSGA-II

Sadržaj

Poglavlje 1

Uvod

Suvremeni svijet nezamisliv je bez brojnih uređaja i sustava upravljanih računalom. Od upravljanja ciklusom pranja rublja u suvremenoj perilici, preko komunikacije pametnim telefonima pa sve do navigacije zrakoplova i raketa - računala čine srž modernih potrošačkih, civilnih, industrijskih i vojnih uređaja i sustava. U većini ovakvih sustava koriste se tzv. *ugradbena računala* koja se po arhitekturi razlikuju od osobnih ili poslužiteljskih računala. Ugradbena računala su potpuno integrirana u sustav, a pri njihovoj izgradnji se ne koriste klasične opće-namjenske konfiguracije nego se komponente posebno odabiru s obzirom na ciljanu vrstu aplikacije. Time se postižu bolje performanse za nižu cijenu. Osim cijene, optimira se veličina, potrošnja energije kod baterijski napajanih uređaja, a u nekim slučajevima i razina zagrijavanja kada ne postoji aktivno hlađenje (npr. mobilni uređaji).

Porast složenosti aplikacija za koje se koriste ugradbeni sustavi zahtijeva da svaka sljedeća generacija implementira sve naprednije i raznovrsnije funkcionalnosti. Prema posljednjem izvješću *ITRS - International Technology Roadmap for Semiconductors* ključni aspekt u razvoju budućih mobilnih i ugradbenih sustava bit će postizanje eksponencijalno rastućih računalnih performansi uz zadržavanje količine napora uloženog u njihov razvoj na razini jednakoj današnjoj [1]. Postizanje zadanih performansi, uz visoku energetsku učinkovitost i nisku cijenu, zahtijeva korištenje višeprocesorskih sustava na čipu - MPSoC (engl. *multiprocessor system-on-chip*) koji omogućavaju paralelnu obradu podataka i upravljanje. Takvi sustavi sadrže dvije ili više različitih vrsta procesora te barem dvije vrste memorija preko kojih ti procesori razmjenjuju podatke, ne računajući pritom priručne memorije. Platforma može sadržavati više instanci iste vrste procesora i memorije. Tipično se koriste procesori arhitektura RISC i DSP koji mogu biti implementirani kao zasebni čipovi ili unutar FPGA te DRAM i FPGA BRAM memorije.

Oblikovanje MPSoC sustava predstavlja veliki izazov. Prije svega broj i raznolikost komponenti za sobom povlači značajno povećanje prostora oblikovanja tj. broja mogućih izbora pri odabiru konačnog rješenja. Izgradnja prototipa ili simulacijske okoline za ispitivanje rada svih mogućih kombinacija nije vremenski ni novčano isplativa. Također, komponente od različitih proizvođača zahtijevaju različite alate za oblikovanje i programiranje. Upravo stoga je nužno cjelokupni sustav predstaviti modelima visoke razine apstrakcije na kojima se mogu provoditi automatizirani postupci pretrage prostora oblikovanja. Na taj način moguće je u vrlo ranoj fazi dobiti inicijalnu procjenu performansi i tako suziti prostor oblikovanja. Time se gotovo beskonačan skup mogućnosti različitih konfiguracija svodi na svega nekoliko kandidatskih rješenja koja se mogu zatim implementirati na razini prototipa i detaljno ispitati.

1.1 Motivacija

Kombiniranjem različitih vrsta procesnih, memorijskih i komunikacijskih elemenata, tj. korištenjem heterogenog MPSoC sustava mogu se postići značajna poboljšanja performansi u odnosu na homogeni sustav s jednakim brojem istovjetnih elemenata. Heterogenost sustava se, osim u različitoj brzini rada procesora i kapacitetu memorija, prije svega očituje u arhitekturi procesnih, memorijskih i komunikacijskih elemenata od kojih je izgrađen sustav što znači da svi elementi nisu jednako pogodni za svaku vrstu zadaće. Stoga je za učinkovito iskorištavanje heterogenog sustava nužno razdijeliti aplikaciju u manje cjeline (npr. procedure) i za te manje cjeline promatrati performanse izračuna (engl. *computation*) i komunikacije (engl. *communication*) na različitim elementima od kojih se gradi heterogeni sustav. Na taj način može se odrediti koji elementi su pogodniji za koje dijelove aplikacije i postići optimalno raspoređivanje na prikladne procesne, memorijske i komunikacijske elemente.

Zbog eksponencijalno rastuće složenosti heterogenih sustava, procjenjuje se da će se produktivnost razvojnih inženjera morati povećati i do deset puta kako bi se moglo uspješno realizirati zahtjeve u okviru vremenskog i novčanog troška sličnog današnjem [2]. Ključ za povećanje učinkovitosti je donošenje ispravnih odluka koje usmjeravaju proces razvoja u ranoj fazi čime se povećava vjerojatnost zadovoljavanja danih zahtjeva na sustav u okviru predviđenog vremena i troška, izbjegava naknadne izmjene i popravke na sustavu te smanjuje rizik.

Tradicionalne metodologije razvoja ne pružaju prikladnu podlogu za razvoj heterogenih višeprocesorskih sustava te ih postupno zamjenjuju nove metode i alati. Oni omogućavaju podiza-

nje razine apstrakcije u gotovo svim fazama procesa oblikovanja što vodi učinkovitijoj uporabi dostupnih resursa. Glavna područja istraživanja u polju oblikovanja heterogenih višeprocesorskih računalnih sustava su:

1. rana procjena performansi kao npr. vrijeme izvođenja, potrošnja energije, zauzeće prostora i cijena,
2. učinkovitost pretrage prostora oblikovanja.

Uz kvalitetu dobivenih rješenja, veliki naglasak se stavlja i na brzinu kojom se dobivaju rješenja. Dosadašnja postignuća na tom području su prvenstveno u vidu podizanja razine apstrakcije što omogućava odvijanje pretrage u vrlo ranoj fazi oblikovanja. Postignuti su značajni rezultati u vidu gotovih metodologija i alata prikladnih za korištenje. Manjkavosti trenutno dostupnih alata i metodologija očituju se u još uvijek slabo automatiziranim, vrlo složenim i vremenski zahtjevnim postupcima, kojima nedostaje modularnost i mogućnost ponovne iskoristivosti rezultata što bi pridonijelo optimizaciji vremena i troška razvoja.

Ovaj doktorat bavi se upravo tematikom pretrage prostora oblikovanja heterogenih višeprocesorskih platforma. Predlaže se metoda pretrage prostora oblikovanja koja obuhvaća postupak rane procjene trajanja izvođenja te heuristiku za rješavanje optimizacijskog problema raspoređivanja u heterogenim MPSoC sustavima. Rana procjena trajanja izvođenja aplikacije temelji se na konceptu *elementarnih operacija*. Elementarne operacije su novi koncept, također izložen u sklopu ovog doktorata, a predstavljaju dijelove izvornog kôda koji omogućavaju određivanje trajanja izvođenja pojedinih operacija na različitim konfiguracijama platforma bez izrade modela procesorskog podatkovnog puta i priručne memorije. Korištenjem elementarnih operacija izgrađuje se apstraktni model aplikacije i višeprocesorske platforme. Ti modeli se koriste u heurističkoj metodi pretrage prostora oblikovanja koja se temelji na postupcima iz područja evolucijskog računarstva uz prilagodbu specifičnostima heterogenih MPSoC sustava. Prednost pretrage prostora oblikovanja korištenjem apstraktnih modela jest smanjenje vremena i napora potrebnih za evaluaciju potencijalnih rješenja što značajno ubrzava proces pretrage. Heurističke metode optimizacije iz evolucijskog računarstva omogućavaju optimizaciju više kriterija istovremeno, iterativno poboljšavaju pronađena rješenja i vrlo se dobro mogu prilagoditi radu s apstraktnim modelima kakvi se koriste za modeliranje heterogenih višeprocesorskih sustava. U predloženoj metodi, pri raspoređivanju dijelova aplikacije na elemente platforme, optimiziraju se izračun i komunikacija istovremeno prema dva kriterija: vremenu izvođenja i zauzeću. Predloženu metodu karakteriziraju modularnost, skalabilnost i ponovna uporabivost čime se postiže

smanjenje jaza između brzine dobivanja potencijalnih rješenja i točnosti procjene njihovih performansi.

1.2 Pristup i doprinos

Doktorsko istraživanje provedeno je u tri faze. U prvoj fazi u fokusu istraživanja je bila problematika procjene trajanja izvođenja aplikacije analizom isključivo na razini izvornog kôda. U najuspješnijim radovima do sada ostvarena je vrlo visoka razina preciznosti procjene s najvećom pogreškom od oko 10% [3, 4, 5, 6, 7]. Sve metode prikazane u spomenutim radovima temelje se na analizi aplikacije na razini osnovnih blokova izvornog kôda (engl. *basic block*). Trajanje izvođenja osnovnih blokova se određuje pomoću simulatora na razini instrukcija čija je pogreška u procjeni oko 1%, a zatim se pomoću apstraktnih modela aplikacije i sklopovlja računski određuje procjena ukupnog trajanja izvođenja. Ovakav pristup zahtijeva ulaganje velikog početnog napora uz malu do nikakvu ponovnu iskoristivost dobivenih rezultata za neku drugu aplikaciju i/ili platformu. Slaba ponovna iskoristivost je posljedica odabira osnovnih blokova kao najmanje količine kôda za koju se mjeri trajanje izvođenja budući da se osnovni blok rijetko pojavljuje u istom obliku u različitim aplikacijama.

Kao pokušaj nadogradnje dosadašnjih istraživanja s ciljem povećanja ponovne iskoristivosti rezultata, u ovom doktorskom radu je predložen koncept *elementarnih operacija*. Ovaj koncept se temelji na ideji da se operacije - sintaksne jedinice programskog jezika koje se promatraju kao cjeline (engl. *statement, code statement*), klasificiraju u elementarne operacije s obzirom na vrstu operatora i operanada koje u sebi sadrže te se svaka elementarna operacija pri procjeni trajanja izvođenja promatra zasebno. U radu je predložen skup elementarnih operacija koji se sastoji od podskupova: cjelobrojne operacije, operacije pomičnog zareza, logičke i memorijske operacije. Podjela elementarnih operacija u navedene podskupove je odraz strukture podatkovnog puta izvršne sklopovske arhitekture čime se na implicitan način odražavaju karakteristike arhitekture sklopovlja. Temeljne značajke ovog koncepta su granuliranost analize dijelova kôda na razini koja omogućava ponovnu iskoristivosti uz prosječnu pogrešku procjene manju od 10%.

Tijekom prve faze istraživanja analizirana je pogodnost koncepta elementarnih operacija za procjenu trajanja izvođenja dijelova aplikacije na heterogenoj platformi te moguća poboljšanja i nadogradnje koncepta. Izrađeni su ispitni slučajevi za mjerenje trajanja izvođenja elementarnih operacija na sklopovskim platformama s različitim procesnim elementima i pripadajućim

memorijskim konfiguracijama. Mjerenja su provedena na sklopovskim platformama koje su predstavnici tipičnih heterogenih višeprocesorskih ugradbenih sustava:

- razvojna ploča Xilinx Zynq ZC706 [8]
- razvojna ploča Adapteva Parallella [9].

U drugoj fazi razvijena je metoda za procjenu trajanja izvođenja aplikacije primjenom elementarnih operacija. Metoda se sastoji od faze analize i faze procjene. U fazi analize izrađuje se profil platforme i aplikacije. Platforma se profilira izvođenjem posebno osmišljenog skupa mjernih programa za elementarne operacije *ELOPS* (engl. *ELementary OPerationS Benchmark*) na svim elementima platforme, čime se dobiva stvarno trajanje izvođenja elementarnih operacija za svaki dio platforme. Profil aplikacije je transformacija izvornog C kôda aplikacije u listu elementarnih operacija strukturiranih u petlje, grane i nizove. U fazi procjene predložen je algoritam pomoću kojeg se na temelju profila aplikacije i platforme dobiva procjena trajanja izvođenja aplikacije na svakom pojedinom procesnom elementu platforme. Metoda je evaluirana na ispitnim aplikacijama koje se vrlo često koriste u ugradbenim sustavima: algoritam kompresije slika JPEG [10] te algoritam enkripcije AES [11].

U trećoj fazi razvijena je metoda za pretragu prostora oblikovanja koja za evaluaciju potencijalnih rješenja koristi procjenu trajanja izvođenja aplikacije primjenom elementarnih operacija. Prema načelu oblikovanja na razini sustava, najprije je predložen apstraktni model sustava koji se sastoji od modela aplikacije i od modela platforme, čime se razdvaja funkcionalnost od strukture te komunikacija od računanja. Aplikacija je na visokoj razini modelirana korištenjem elemenata teorije grafova pomoću kojih se modelira tok izvođenja aplikacije i međuzavisnosti između pojedinih dijelova aplikacije. Svaki čvor u grafu karakteriziran je na temelju koncepta elementarnih operacija. Model platforme također se opisuje pomoću elemenata iz teorije grafova za prikaz topologije sustava, a svaki element je karakteriziran pomoću profila dobivenog izvođenjem mjernih programa ELOPS. Sama metoda pretrage prostora oblikovanja, tj. mogućih pridruživanja (engl. *mapping*) dijelova aplikacije na dijelove platforme, razvijena je na temelju evolucijskog algoritma za višekriterijsku optimizaciju NSGA-II [12] te posebno prilagođena kako bi radila sa zadanim modelom sustava i bila u mogućnosti riješiti specifične probleme koji se pojavljuju u heterogenoj višeprocesorskoj okolini kao što su razdvojenost komunikacije i računanja, nepotpuna povezanost elemenata platforme i sl.

Doprinosi ovog rada su:

1. Klasifikacija izvornog kôda aplikacije na temelju koncepta elementarnih operacija.

2. Model aplikacije i sklopovske platforme za analitičku procjenu trajanja/vremena izvođenja.
3. Metoda pretrage prostora oblikovanja postupcima višekriterijske optimizacije.

pri čemu su svi dobiveni rezultati eksperimentalno provjereni i potvrđeni.

1.3 Struktura rada

U poglavlju koje slijedi ukratko su opisane glavne značajke oblikovanja višeprocesorskih sustava na čipu. Definirane su razine apstrakcije koje se koriste u različitim metodologijama oblikovanja te opisan odnos između njih. Nadalje su opisane i uspoređene različite metodologije razvoja ugradbenih sustava. Analizirane su prednosti i nedostaci tradicionalnih metodologija te ukratko prodiskutiran način kako je iz njih proizašla suvremena metodologija oblikovanja na razini sustava i koji su izazovi na koje ona treba odgovoriti.

U trećem poglavlju je opisana problematika rane procjene trajanja izvođenja aplikacije na razini izvornog kôda te dan pregled dosadašnjih radova iz tog područja. Predložena je nova metoda za procjenu trajanja izvođenja pomoću klasifikacije izvornog kôda aplikacije na temelju koncepta elementarnih operacija. U poglavlju je detaljno opisana predložena klasifikacijska shema te postupak procjene rezultata. Također su izloženi rezultati provedenih eksperimenata.

Četvrto poglavlje obrađuje problematiku pretrage prostora oblikovanja. Na samom početku su izneseni osnovni pojmovi i definicije, zatim je dan pregled postojećih radova. U sklopu tog poglavlja predstavljen je novi model aplikacije i sklopovske platforme za analitičku procjenu trajanja izvođenja izgrađen na načelima oblikovanja na razini sustava i koncepta elementarnih operacija. Objašnjeni su koncepti definiranja modela aplikacije i platforme iz kojih proizlaze njihove specifikacije. Zatim je prezentirana nova metoda pretrage prostora oblikovanja postupcima višekriterijske optimizacije. Na kraju poglavlja su izloženi rezultati provedenih eksperimenata te provedena diskusija.

Poglavlje 2

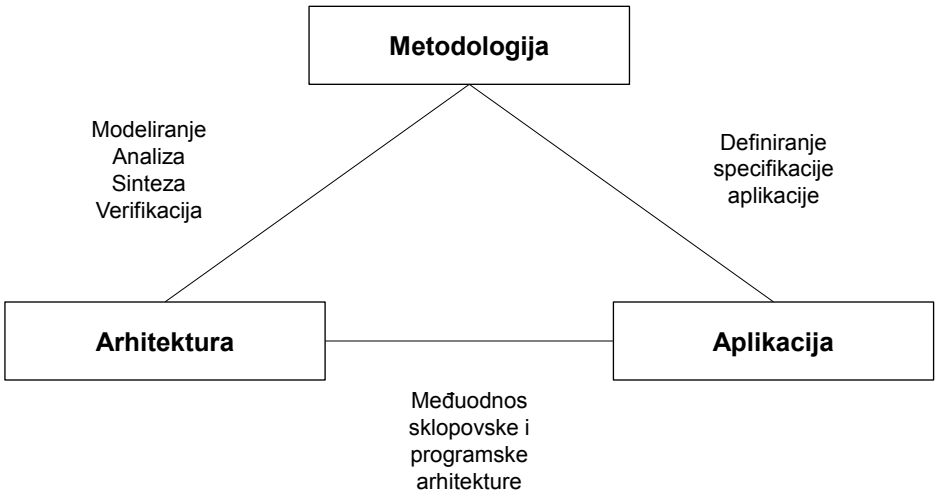
Oblikovanje višeprocesorskih sustava na čipu

Oblikovanje ugradbenih sustava je proces definiranja arhitekture, komponenata, modula, sučelja i podataka u sustavu koji zadovoljavaju zadane specifikacije [13]. To je složen proces koji se razlaže u faze od kojih se neke izvode slijedno, a neke ponavljaju u iteracijama. Moderni sustavni pristup oblikovanju ugradbenih sustava podrazumijeva slijed koraka koji postupno rafiniraju visoki stupanj apstrakcije prema nižem. Proces počinje s inicijalnom specifikacijom sustava te se nadalje razmatraju različita moguća rješenja, iterativno izvodeći implementaciju, verifikaciju i poboljšavanje sve do konačnog rješenja - konkretnog proizvoda. Pri tome inicijalna specifikacija sustava određuje prostor potencijalnih rješenja te metodu i alate kojima će se pretražiti [14].

Da bi učinkovito ostvarili različite funkcionalnosti koje se zahtijevaju od modernih ugradbenih sustava, nužno je koristiti različite vrste procesnih, memorijskih i komunikacijskih elemenata, tj. heterogenu arhitekturu platforme [15]. U takvom okruženju, pri razvoju sustava nužno je razmotriti slijedeća tri aspekta oblikovanja [16]:

1. *arhitektura* – podrazumijeva arhitekturu sklopovlja i aplikacije te odnos između njih,
2. *aplikacija* – odnosi se na dobro razumijevanje zadatka za koji se gradi sustav te omogućuje uvođenje optimizacija i sprječava slijepe ulice u oblikovanju,
3. *metodologija* – obuhvaća faze u procesu oblikovanja te alate i resurse koji se pri tome koriste.

Odnos arhitekture, aplikacije i metodologije je ilustriran na slici 2.1. Posebnost ugradbenih sustava je nužnost istovremenog oblikovanja i prilagodbe sklopovlja i aplikacije kako bi se nji-



Slika 2.1: Aspekti oblikovanja modernih ugradbenih sustava [16]

hovom kombinacijom postigle što bolje performanse. Dubinska analiza i poznavanje aplikacije za koju se radi sustav kao i karakteristika sklopovskih komponenti dostupnih pri izradi platforme važan su preduvjet za izradu optimalnog sustava. Pri tome ne postoji jedan standardni postupak nego je često potrebno kombinirati i prilagođavati postojeće metodologije zadanom problemu. Dobra razrada arhitekture, kako sklopovlja tako i aplikacije, te razumijevanje odnosa između njih temelj je procesa oblikovanja. Po pitanju sklopovlja najvažniji je odabir odgovarajuće arhitekture procesora za određenu vrstu aplikacija, a po i potrebi oblikovanje potpuno nove vrste sklopovskog akceleratora za pojedine zadaće ukoliko klasični procesori ne daju dovoljno dobre performanse (engl. *co-design*). S aspekta aplikacije važno je dobro razumijevanje zadatka koji se obavlja. To omogućava uvođenje optimizacija, pronalaženje i iskorištavanje paralelizma u aplikaciji te sprječava slijepe ulice u oblikovanju.

Metodologija podrazumijeva faze u procesu oblikovanja te alate i resurse koji se pri tome koriste. Uloga metodologije oblikovanja je iznimno važna budući da dobrim odabirom osigurava postizanje zadanih performansi i visoke pouzdanosti rada konačnog produkta te optimizira vrijeme i trošak oblikovanja. Posebnost metodologije oblikovanja modernih ugradbenih sustava je složenost i varijabilnost iz razloga što se svaki novi ugradbeni sustav prilagođava zasebnom, novom problemu. Osnovni elementi svake metodologije su analiza i simulacija u ranoj fazi kako bi se što ranije mogle donijeti ključne odluke koje dalje usmjeravaju proces oblikovanja. Pri tome važnu ulogu ima automatizacija cijelog procesa kroz različite alate. Nove metodo-

logije oslanjaju se na modele za savladavanje složenosti i omogućavanje donošenja odluka u ranim fazama.

Veličina i nepravilnost prostora oblikovanja modernih ugradbenih sustava predstavlja veliki izazov pri predviđanju vremena i resursa potrebnih za pronalaženje najboljeg rješenja. Trenutno je uočljiv rastući jaz između složenosti novih sustava temeljenih na heterogenim arhitekturama i produktivnosti inženjera koji rade na oblikovanju takvih sustava [17]. Mnogo je radova napisano i mnogo posla napravljeno kako bi se odgovorilo na ovaj izazov, no još uvijek ne postoji jedinstvena metodologija koja bi ponudila cjelovito rješenje.

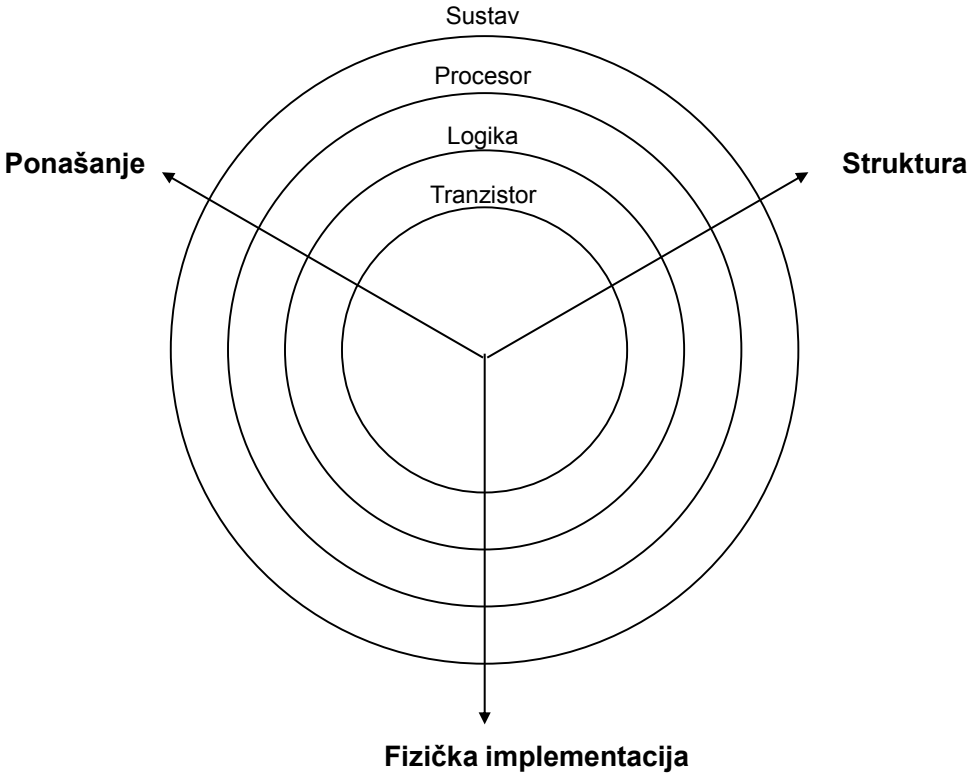
Nova generacija metodologija za oblikovanje sustava stavlja naglasak na ranu procjenu performansi potencijalnih rješenja te ubrzanje pretrage prostora oblikovanja. Pri tome je ključ u povišenju razine apstrakcije u ranim fazama oblikovanja koja omogućuje brzu pretragu prostora dizajna i procjenu performansi. Veliki naglasak se stavlja također i na ponovnu iskoristivost rješenja kako bi se dodatno smanjili troškovi, vrijeme i napor koji se ulažu u razvoj i ispitivanje konačnog proizvoda (sustava). Metodologija bazirana na platformi PBD (engl. *platform-based design methodology*) proizlazi kao trenutno najučinkovitiji pristup [18].

U nastavku poglavlja iznesene su ključne značajke modernog pristupa razvoju ugradbenih sustava i kako pojedine metodologije odgovaraju na postojeće izazove.

2.1 Razine apstrakcije u oblikovanju sustava

Odnos između razina apstrakcije u različitim metodologijama oblikovanja moguće je slikovito prikazati korištenjem tzv. *Y-Chart* modela, razvijenog 1983. godine [13, 19]. Osnovna ideja ovog pristupa jest da se aplikacija koja predstavlja funkcionalnost koju sustav obavlja i arhitektura sustava modeliraju zasebno, a zatim se dijelovi aplikacije raspodjeljuju po komponentama u arhitekturi. Budući da je za ne-trivijalne probleme moguć iznimno velik broj različitih pridruživanja (engl. *mapping*), taj se proces ponavlja na različitim razinama apstrakcije sve dok se ne dođe do zadovoljavajućeg rješenja, pri čemu se mogu također mijenjati i poboljšavati parametri aplikacije i arhitekture [20].

Na slici 2.2 je prikazan Y-Chart model s tri osi koje predstavljaju tri komponente oblikovanja: ponašanje tj. funkcionalnost, strukturu u obliku net-liste ili blok dijagrama i fizičku implementaciju sustava na ploči. Ponašajna komponenta se promatra kao crna kutija s jasno definiranim ulazima i očekivanim izlazima. Struktura podrazumijeva skup komponenti arhitekture



Slika 2.2: Y-Chart model [13]

sustava i njihovu povezanost. Fizički sustav predstavlja parametre konkretne implementacije kao što su veličina i položaj na fizičkoj platformi koja može biti jedan čip ili cijela razvojna ploča.

Razine apstrakcije u Y-Chart modelu su predstavljene koncentričnim kružnicama. Uglavnom se koriste četiri razine: fizička (engl. *circuit*), logička (engl. *logic*), procesorska i razina sustava. Na fizičkoj razini se govori o tranzistorskim ćelijama. Na logičkoj razini definirani su logički sklopovi (engl. *logic gates*) i bistabili pomoću kojih se ostvaruju komponente za obradu i spremanje podataka. Na procesorskoj razini se razmatraju standardni procesori ili posebno oblikovani namjenski sklopovi za izvođenje točno određenih vrsta zadataka te različiti upravljačia na sabirnicama, memorije i sl. Razina sustava obuhvaća kompletan sustav u kojem međudjeluju procesni, memorijski i komunikacijski elementi.

Na svakoj razini apstrakcije moguće je definirati tri modela za svaku komponentu: ponašajni, strukturni i fizički. No, najčešće to nije potrebno pa se u modernim metodologijama za tri najviše razine apstrakcije definiraju ponašajni modeli koji u sebi sadrže parametre za procjenu

performansi kao npr. brzine rada, kašnjenja, potrošnje energije, troška, pouzdanosti, veličine i sl., a tek se na razini logičkih sklopova definira detaljna strukturna i fizička specifikacija.

2.2 Metodologije oblikovanja sustava

Prema redoslijedu prolaska po razinama apstrakcije na modelu *Y-Chart* razlikuju se dvije tradicionalne metodologije oblikovanja: odozdo prema gore (engl. *bottom-up*) i odozgo prema dolje (engl. *top-down*). Ove dvije metodologije imaju dobro razrađen i opisan slijed izvođenja i semantiku te se mogu upotrijebiti za rješavanje različitih problema u oblikovanju. No, s obzirom na porast složenosti ugradbenih sustava nove generacije i zahtjeva za skraćenjem vremena do izlaska na tržište (engl. *time to market*), ove metode ne mogu u potpunosti udovoljiti zahtjevima. Nova metodologija oblikovanja na razini sustava - SLD (engl. *system-level design*) koja se ponekad još naziva i oblikovanje temeljeno na platformi - PBD (engl. *platform-based design*) spaja pozitivne aspekte obje metodologije i nalazi nova rješenja za postojeće probleme kako bi ponudila odgovarajuće rješenje.

U nastavku poglavlja detaljnije su opisane karakteristike svih triju metodologija.

2.2.1 Metodologija odozdo prema gore

Najstarija metodologija oblikovanja ugradbenih sustava je metodologija odozdo prema gore (engl. *bottom-up*). Za nju je karakteristično da se najprije izgrađuju dijelovi koji se zatim povezuju u sustav. Kretanje po razinama apstrakcije je od najnižeg (fizičkog) prema najvišem (sustav).

Prednost ove metodologije jest strogo razdvajanje razina apstrakcije koje su potpuno definirane ponašajno, strukturno i fizički. Na ovaj način moguće je izvoditi globalo-distribuirano oblikovanje na svakoj razini jer se na svakoj razini isporučuje gotov proizvod. Nedostatak ovog pristupa jest što je potrebno na nižoj razini proizvesti sve moguće komponente sa svi mogućim parametrima podešavanja kako bi se udovoljilo sadašnjim i budućim zahtjevima aplikacija na višoj razini. Ovo je u praksi gotovo nemoguće ostvariti pa je često nužan ponovni razvoj ispočetka.

2.2.2 Metodologija odozgo prema dolje

Proces oblikovanja u metodologiji odozgo prema dolje (engl. *top-down*) počinje s ponašajnim modelom iz kojeg se generira platforma sa zadanim parametrima pojedinih komponenti, bez definiranja strukture i rasporeda. Izrada pojedinačnih komponenti na nižoj razini počinje tek kada je oblikovanje na višoj razini u potpunosti dovršeno. Na nižim razinama apstrakcije komponente se razlažu u logičke i fizičke komponente koje zadovoljavaju specifikacije više razine. Na kraju se sve komponente fizički implementiraju na čipu ili integriranoj fizičkoj platformi.

Ova metoda je bila vrlo često korištena u razvoju ranih računala, no za današnje je sustave postupak previše složen budući da iz same funkcionalne specifikacije nije moguće odrediti performanse konačnog sustava. Iz tog razloga je i vrlo teško napraviti učinkovitu optimizaciju sustava te je potreban velik broj ponavljanja iteracija u oblikovanju i implementaciji kako bi se došlo do valjanog rješenja, a iznimno je lako zaglaviti u slijepoj ulici.

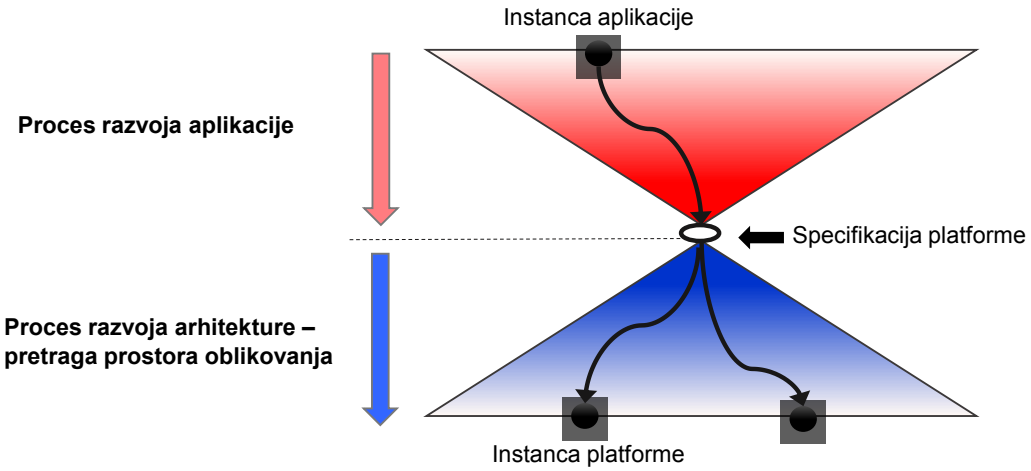
2.2.3 Oblikovanje na razini sustava

U posljednjem desetljeću prvenstvo preuzima nova metodologija pod nazivom oblikovanje na razini sustava SLD (engl. *system-level design*) koja nastoji premostiti jaz između faze specifikacije te faze oblikovanja i implementacije sustava koji postoji u tradicionalnim metodologijama [21]. Iako ne postoji zajednički konsenzus o definiciji ove metodologije, SLD se najčešće predstavlja kao skup različitih metoda i pristupa temeljenih na ideji o nužnosti povećanja razine apstrakcije radi skraćivanja vremena pretrage potencijalnih rješenja te automatizacije procesa sinteze prema nižim razinama kako bi se premostio implementacijski jaz [22]. Ova metodologija objedinjava najbolje aspekte dviju ranije spomenutih tradicionalnih metodologija te pokušava razriješiti njihove nedostatke kroz sustavan pristup razvoju. Metodologija SLD funkcionira na razini iznad RTL-a (engl. *register-transfer čevel*), a ključni koncept je specifikacija na razini sustava. Ona uključuje specifikaciju arhitekture na razini sustava i osnovu ponašajnu specifikaciju aplikacije. Time se definira ukupan prostor oblikovanja i mogućih konačnih rješenja. Proces razvoja je iterativan te se kroz slijed koraka (implementacija, verifikacija i redefiniranje specifikacije) početna specifikacija sustava, zadana na visokoj razini apstrakcije, razrađuje i postupno transformira u konačno rješenje [23].

Oblikovanje temeljeno na platformi PBD (engl. *platform-based design*) je jedan od najistaknutijih pristupa SLD-u. Platforma kao koncept postoji već duže vrijeme te podrazumijeva postojanje knjižnice komponenata (engl. *component library*) koje se po potrebi mogu dodavati

ili uklanjati iz dizajna kako bi se prilagodilo zahtjevima sustava. Ključni element ovog pristupa je ponovna iskoristivost postojećih elemenata koji se kombiniraju na različit način, ovisno o potrebi. Time se bitno skraćuje vrijeme razvoja i smanjuje trošak. Ova metodologija oblikovanja se može ukratko opisati kao nalaženje na sredini (engl. *meet-in-the-middle*) tj. kompromis dviju tradicionalnih metodologija. Pristup odozgo prema dolje se primjenjuje kod raspodjele dijelova aplikacije na dijelove platforme uz propagaciju ograničenja, dok se pristup odozdo prema dolje primjenjuje na nižim razinama apstrakcije tako što se platforma gradi koristeći gotove komponente iz biblioteke (engl. *library*).

Srž metodologije PBD jest korištenje različitih razina apstrakcije kako bi se napravila raspodjela funkcionalnosti na arhitekturu tj. elemente platforme kao što je ilustrirano na slici 2.3 izrađenoj na temelju [22]. Posebna se pažnja obraća na podjelu oblikovanja na sklopovski i programski dio na višim razinama apstrakcije kako bi se ograničio prostor potencijalnih rješenja i eliminiralo velike iteracije u petlji u procesu daljnje razrade. Postoje dvije jasne faze u razvoju predstavljene s dva trokuta na slici. Gornji trokut predstavlja fazu razvoja aplikacije. Nakon što je definirana aplikacija i njena ponašajna specifikacija, izrađuje se model arhitekture na najvišoj razini apstrakcije - tzv. *specifikacija platforme*, na temelju koje je moguće dati osnovne procjene trajanja izvođenja, zauzeća, potrošnje energije i sl. Komunikacija (engl. *communication*) i izračun (engl. *computation*) su strogo odvojeni, a svaki element platforme karakterizira skup parametara kojima su opisane njegove performanse na visokoj razini. Donji trokut predstavlja fazu razvoja arhitekture (tzv. pretraga prostora oblikovanja). U toj fazi početni model platforme se razrađuje prema nižim i sve konkretnijim razinama tražeći stvarna rješenja i provjeravajući da ona zadovoljavaju početna zadana ograničenja. Konačna implementacija se dobiva sintezom iz modela više razine te se po potrebi još ručno poboljšava i dorađuje. Potrebno je naglasiti da su sva moguća rješenja zapravo samo različite kombinacije elemenata iz knjižnice komponenata platforme, otkuda potječe i naziv metodologije.



Slika 2.3: SLD - PBD proces [22]

Ukratko, glavna načela PBD-a su:

1. oblikovanje počinje s modelom na visokoj razini apstrakcije,
2. prostor oblikovanja (mogućih rješenja) je ograničen dostupnih skupom komponenti,
3. oblikovanje se izvodi kao postupak u kojem se inicijalna specifikacija na visokoj razini apstrakcije u koracima razrađuje na sve nižu razinu apstrakcije, sve dok se ne dođe do konačne implementacije.

Ovdje je nužno spomenuti da neki autori rade razliku između PBD-a i SLD-a i ne prihvaćaju PBD kao dio SLD-a budući da koncept platforme postoji odavno i ne mora nužno u sebi obuhvaćati specifikaciju i modeliranje na visokoj razini [13].

Zaključno, SLD, tj. PBD kao jedan od pristupa, ne rješava u potpunosti sve probleme oblikovanja suvremenih heterogenih višeprocesorskih sustava. Izazovi na koje se još treba dati odgovor su:

1. učinkovito savladavanje heterogenosti i složenosti sklopovske platforme do kojeg nužno dolazi kako bi se udovoljilo zahtjevima na pouzdanost, potrošnju energije i performanse uz minimiziranje troška,
2. povećanje složenosti aplikacija koje se izvode na ugradbenim sustavima i koje se moraju posebno prilagoditi kako bi radile na specijaliziranom sklopovlju (suprotno od općenamjenskih komercijalnih aplikacija koje se izvode na osobnim računalima),
3. složenost integracije aplikacije i platforme zahtijeva bolje razumijevanje međudjelovanja podsustava koji čine cjelinu i to već u ranoj fazi donošenja odluka.

U savladavanju ovih izazova buduće metodologije mogu kao svoj temelj preuzeti visoku razinu

apstrakcije, glavnu karakteristiku SLD-a. Naime, modeli visoke razine apstrakcije skrivaju složene mehanizme interakcije aplikacije i arhitekture čime direktno smanjuju složenost procesa pretrage potencijalnih rješenja i ubrzavaju ga. Ključni element je izgradnja adekvatnog modela koji će na visokoj razini reprezentirati značajke heterogene arhitekture na način koji omogućava točnu procjenu performansi i usporedbu potencijalnih rješenja.

Poglavlje 3

Procjena trajanja izvođenja na razini izvornog kôda

Moderni ugradbeni sustavi temelje se na sustavima na čipu SoC (engl. *systems on chip*) heterogene strukture procesnih, memorijskih i komunikacijskih elemenata. Na taj način postižu se visoke performanse uz minimalnu potrošnju energije, manje zauzeće prostora i nižu cijenu. Upravo stoga oblikovanje heterogenih sustava je znatno složenije nego oblikovanje homogenih sustava: trenutni stupanj rasta složenosti oblikovanja je eksponencijalan [1] a procjenjuje se kako će se produktivnost razvojnih inženjera morati povećati i do deset puta kako bi se zadovoljili svi zahtjevi unutar zadanih vremenskih i novčanih ograničenja [2] .

Ključ uspješnog procesa razvoja je u donošenju kvalitetnih odluka koje usmjeravaju proces u ranoj fazi, prije izgradnje prvog prototipa. Odluke se donose prilikom pretrage prostora oblikovanja - DSE (engl. *design space exploration*) tako što se ispituju performanse aplikacije na različitim konfiguracijama platforme. DSE u je heterogenim sustavim iznimno važan budući da heterogenost elemenata platforme znači da različiti procesni elementi neće biti jednako učinkoviti pri izvođenju različitih vrsta aplikacija, niti će svaka konfiguracija memorijskih i komunikacijskih elemenata biti jednako pogodna za različite tipove aplikacija. Pri ispitivanju performansi potrebno je imati metode pomoću kojih je moguće što ranije, uz što manji uloženi napor i vrijeme te s prihvatljivom razinom pogreške dati procjenu performansi sustava. Pri tome se najveći naglasak stavlja na vremensku dimenziju, tj. procjenu trajanja izvođenja aplikacije.

U ovom doktorskom radu predlaže se metoda za ranu procjenu trajanja izvođenja na temelju izvornog kôda aplikacije (engl. *source-level*) pod nazivom *ELOPS-EM* (*ELementary OPerationS based Estimation Method*). U osnovi metode ELOPS-EM je klasifikacija različitih vrsta

operacija - sintaksnih jedinica koje se promatraju kao cjelina u izvornom kôdu aplikacija pisanih u jeziku C, po standardu C11, u *elementarne operacije*. Na taj način identificiraju se zasebni dijelovi izvornog kôda čije se trajanje izvođenja na platformi može mjeriti potpuno neovisno o drugim dijelovima kôda i samoj aplikaciji u kojoj se pojavljuju. Temeljem tih mjerenja može se dati rana procjena trajanja izvođenja cijele aplikacije bez potrebe za simulacijom na razini osnovnih blokova i bez izrade matematičkog modela podatkovnog puta i priručne memorije procesora, kakve zahtijeva većina ranije spomenutih pristupa.

Postupak procjene trajanja izvođenja se izvodi u dvije faze: *analiza* i *procjena*. Tijekom analize izrađuju se profili svih aplikacija i platformi koje se uzimaju u razmatranje. Aplikacija se profilira tako što se u izvornom kôdu aplikacije identificiraju sve vrste prisutnih elementarnih operacija te strukture u kojima se one nalaze: petlje, granjanja ili nizovi operacija iste vrste. Platforma se profilira izvođenjem skupa mjernih programa pod nazivom *ELOPS* zasebno na svakoj konfiguraciji platforme i za svaku razinu optimizacije prevoditelja koja se razmatra. Ovaj skup mjernih programa posebno je izrađen u sklopu doktorskog istraživanja i služi za mjerenje vremena izvođenja elementarnih operacija na stvarnim platformama. Objedinjeni rezultati izvođenja mjernih programa na nekoj konfiguraciji platforme za sve razine optimizacije čine profil platforme za tu konfiguraciju. Profili aplikacije i platforme se trajno spremaju u bazu podataka. U fazi procjene, na temelju profila aplikacije i platforme, daje se procjena trajanja izvođenja aplikacije za zadanu konfiguraciju platforme. Profili aplikacije i platforme su međusobno potpuno nezavisni te je stoga moguće u budućnosti ponovno iskoristiti ranije dobivene profile za različite kombinacije aplikacija i platformi.

U nastavku poglavlja dan je pregled relevantnih radova iz područja, detaljno je opisana predložena metoda ELOPS-EM: postupak klasifikacije elementarnih operacija, profiliranja aplikacije i platforme te algoritam koji na temelju profila računa procjenu trajanja izvođenja. Metoda je evaluirana koristeći aplikacije koje implementiraju algoritme JPEG i AES na nekoliko različitih konfiguracija platforme Xilinx Zynq ZC706.

3.1 Pregled radova

Tradicionalno se za procjenu trajanja izvođenja koristi simulator seta instrukcija - ISS (engl. *instruction set simulator*) na kojem se izvodi cijela aplikacija i koji daje procjenu s oko 1% pogreške u odnosu na fizičku platformu. Iako je razina pogreške prihvatljiva, temeljni nedostatak

korištenja ISS-a jest da za različite platforme treba koristiti različite simulatore. Svaki simulator je zasebni alat i zbog heterogenosti alata je gotovo nemoguće povezati više simulatora u jedinstven automatizirani tok kakav bi bio poželjan u procesu DSE. To znači da svaki put kada se napravi bilo kakva modifikacija u aplikaciji, npr. paralelizacija dijela kôda, odmotavanje petlje i sl., potrebno je za svaki simulator posebno prevesti i pokrenuti aplikaciju i to najčešće ručno. Nadalje, izvođenje aplikacije na ISS-u može trajati i nekoliko redova veličine duže nego na stvarnom sklopovlju.

Iz tog razloga, predlažu se nove metode procjene trajanja izvođenja koje nastoje pojednostaviti, ubrzati i automatizirati proces uvođenjem modela visoke razine apstrakcije [4, 5, 6, 7, 24, 25, 26, 27]. Nove metode omogućavaju mnogo brže dobivanje procjene u odnosu na ISS te pružaju veću skalabilnost rješenja, no uz smanjenu točnost procjene. U mnogobrojnim radovima kroz čitavo desetljeće predložene su različite metode koje se generalno mogu podijeliti u dvije skupine:

1. simulacija izvođenja izvornog kôda sa pozadinskom anotacijom (engl. *back-annotation*),
2. analitički pristup koji se oslanja na razne matematičke modele ili strojno učenje.

Simulacijske metode pružaju veću točnost rezultata u odnosu na analitičke metode uz manji stupanj ponovne iskoristivosti i skalabilnosti rješenja, zbog čega zahtijevaju veći uloženi napor i više vremena za dobivanje procjene.

Skup alata za oblikovanje heterogenih sustava na FPGA čipu pod nazivom *Daedalus* uključuje i alat za pretragu prostora oblikovanja pod nazivom *Sesame* [28]. *Sesame* daje ranu procjenu trajanja izvođenja koristeći simulaciju temeljenu na praćenju tragova izvođenja aplikacije (engl. *trace-based*) koji se zatim bilježe u model aplikacije. Razina granulacije je vrlo krupna tako da se bilježe pozivi pojedinih funkcija te pristupi memoriji (u svojstvu komunikacijskog kanala). U skladu s time, model platforme za svaki procesni element ima tablicu trajanja izvođenja pojedinih operacija tj. funkcija. Uz dodatnu kalibraciju pomoću ISS-a, autori navode da se pogreška procjene kreće od 12 do 19% [29].

Alat *MAPS* [30] nudi nekoliko opcija za procjenu trajanja izvođenja. Osnovna opcija je analiza i instrumentacija na razini izvornog kôda. Ona se također temelji na praćenju tragova izvođenja, ali na razini međureprezentacije - IR (engl. *intermediate representation*) i osnovnih blokova - BB (engl. *basic blocks*). Takav pristup je relativno brz, ali prema navodima samih autora može dati pogrešku i do 300%. Naprednija opcija je korištenje dodatnog alata *TotalProf* [31] koji također radi na razini međureprezentacije, ali implementira naprednije op-

cije koje omogućuju simulaciju učinaka cjevovoda i priručne memorije te daje razinu pogreške oko 15%. Ipak za većinu analiza autori koriste virtualnu platformu, sličnu ISS-u, koja iako je najsporija od sve tri opcije, daje rješenja s vrlo malom razinom pogreške od 1 do 2% [30].

Uz ova dva cjelovita alata postoje i pojedinačni radovi koji se bave problematikom rane procjene performansi. Autori u [4, 5, 6, 7, 24, 25, 26, 27], čiji radovi predstavljaju najnovija postignuća u području, pri procjeni trajanja izvođenja koriste označavanje na razini izvornog kôda ili među-reprezentacije. Struktura aplikacije se dijeli u osnovne blokove koristeći prevoditelj GCC, a zatim se pomoću ISS-a određuje trajanje izvođenja svakog bloka. Kasnije se simulira izvođenje aplikacije za različite ulazne vrijednosti koristeći System C i modeliranje na razini transakcija [32]. Pogreške u procjenama se kreću uglavnom ispod 10%. Nedostatak ovakvog pristupa je nemogućnost izravne simulacije učinaka cjevovoda na prijelazima između blokova što zahtjeva dodatnu simulaciju parova blokova za sve moguće kombinacije prethodnik – sljedbenik, a što produljuje vrijeme simulacije. Iako se postiže dobra preciznost procjene trajanja izvođenja, veliki nedostatak predstavlja nemogućnost ponovnog korištenja dobivenih mjerenja.

Radni okvir *DOL* oslanja se na analitičku procjenu koristeći radni okvir *EXPO* [33] u kojem se pomoću dolaznih i uslužnih krivuljama, preuzetih iz RTC-a (engl. *real time calculus*) [34], dobro opisuju periodički događaji i vršna opterećenja kako bi se osiguralo izvođenje u kratkom vremenu. No, za dobivanje preciznih vrijednosti parametara koriste se i druge tehnike poput simulacije na ISS-u.

Nešto drukčiji pristup prikazan je u [25] gdje se koristi analitička procjena temeljena na strojnom učenju. Neuronska mreža (engl. *artificial neural network*) prima broj pojedinih vrsta asemblerskih instrukcija u aplikaciji, za koje je ranije utvrđeno trajanje izvođenja, i na temelju tih podataka daje procjenu trajanja izvođenja cijele aplikacije. Točnost rezultata je nešto niža sa oko 17% pogreške, no sustav je fleksibilniji uz veći stupanj ponovne iskoristivosti rezultata.

Radovi [26] i [27] također se temelje na strojnom učenju. Oni u svom pristupu koriste linearnu regresiju i neuronske mreža te dobivaju nešto veću pogrešku procjene od oko 20%. Radovi [35, 36, 37] predstavljaju hibridne metode u kojima se najprije simulacijom određuje trajanje izvođenja pojedinih funkcija za svaki procesni element zasebno, a zatim se analitičkim metodama dodaju učinci priručne memorije i komunikacije između procesora. Svi navedeni radovi uvode razlikovanje vrsta operacija na razini asemblera ili izvornog kôda.

Pristup izložen u [38], u kojem se procjena trajanja izvođenja računa pomoću linearne re-

gresije, posebno je zanimljiv iz razloga što se za mjerenje trajanja izvođenja instrukcije na virtualnom stroju umjesto standardnih mjernih programa (engl. *benchmarks*) koriste posebno izrađeni mjereni programi. Ti programi sadrže petlje i nizove naredbi pomoću kojih se nastoji izmjeriti utjecaj priručne memorije, cjevovoda i optimizacija koje u kôd uvodi prevoditelj na trajanje izvođenja aplikacije bez izrade konkretnog matematičkog modela arhitekture ili simulacije na razini osnovnih blokova. Međutim, budući da se pristup oslanja na modeliranje na razini međureprezentacije, pojavljuju se problemi kod nekih optimizacija u izvornom kôdu aplikacije kada se virtualne instrukcije ne poklapaju dobro s verzijom koju daje prevoditelj.

Ukratko, simulacijske metode daju rezultate visoke točnosti s razinom pogreške ispod 10%, no zahtijevaju značajno vrijeme i napor za pripremu simulacijske okoline, a ponovna iskoristivost rezultata je mala. Ostale metode koje se temelje na matematičkim modelima i strojnom učenju pokazuju veću ponovnu iskoristivost i skalabilnost dobivenih rezultata uz povećanu razinu pogreške od oko 20%.

3.2 Elementarne operacije

U osnovi svake prethodno opisane metode procjene trajanja izvođenja na razini izvornog kôda je modeliranje aplikacije kao skupa međusobno povezanih manjih jedinica kôda - granula. Granulacija aplikacije u modelu može varirati od vrlo krupne - na razini procedura [28] do vrlo sitne - na razini asemblerskih instrukcija [25]. Krupnija granulacija daje veću točnost, ali manju ponovnu iskoristivost, dok sitnija granulacija ima veće poteškoće s modeliranjem učinaka optimizacije i priručne memorije, ali omogućava veću ponovnu iskoristivost. Mjerenja stvarnog trajanja izvođenja se rade na platformi ili ISS-u samo za granule, a zatim se na temelju mjerenja i modela aplikacije računa procjena ukupnog trajanja izvođenja aplikacije. Izazov je odabrati razinu granulacije i izgraditi model koji će omogućiti što veću ponovnu iskoristivost rezultata uz što manju pogrešku procjene.

Metoda ELOPS-EM (*ELementary OPerationS based Estimation Method*), koja se predlaže u ovom radu, granulira aplikacije u jeziku C na razini operacije kao sintaksne jedinice jezika koje se promatraju kao cjeline (engl. *statement, code statement*). Te sintaksne jedinice se nazivaju *elementarne operacije* i klasificiraju se s obzirom na vrstu operatora i operanada od kojih se sastoje. Prednost ove razine granulacije jest da se mjerenja trajanja izvođenja pojedinih elementarnih operacija mogu provesti neovisno o konkretnoj aplikaciji i onda koristiti za procjenu

u bilo kojoj aplikaciji. Potencijalni problem ove razine granulacije jest uračunavanje utjecaja priručne memorije, cjevovoda i optimizacija na trajanje izvođenja aplikacije. Ovi utjecaji se pojavljuju zbog konteksta u koji su smještene elementarne operacije kao što su npr. petlje i nizovi operacija te se kao rješenje uvodi izrada posebnih mjernih programa koji će u sebi oponašati taj kontekst, slično kao što je napravljeno u [38].

3.2.1 Klasifikacija elementarnih operacija

Detaljna klasifikacija elementarnih operacija radi se prema sljedećoj višerazinskoj shemi. Na najvišoj razini su četiri temeljna skupa operacija koji su vezani uz dijelove arhitekture procesora tipa RISC: cjelobrojne operacije - *INT* (engl. *integer*), operacije s pomičnim zarezom - *FP* (engl. *floating point*), logičke - *LOG* (engl. *logic*) i memorijske - *MEM* (engl. *memory*) operacije.

Sljedeća razina klasifikacije određuje se prema porijeklu operanada u operaciji (tj. lokaciji u memorijskom prostoru na kojoj se pohranjuju) pa se razlikuju operacije s lokalnim (engl. *local*), globalnim (engl. *global*) i operandima koji su parametri procedure (engl. *parameter*). Ovaj kriterij klasifikacije proizlazi iz činjenice da prevoditelj u memoriji različito smješta ove tipove operanada pa se očekuje i različito trajanje izvođenja operacija koje ih sadrže. Operandi koji su definirani lokalno u procedurama smješteni su na stogu te zbog vremenske i prostorne lokalnosti se vrlo vjerojatno nalaze u priručnoj memoriji. S druge strane globalne varijable mogu biti smještene bilo gdje u glavnoj memoriji te je veća vjerojatnost da će se u trenutku pristupa nalaziti izvan priručne memorije što povećava vrijeme dohvata i po nekoliko redova veličine. Parametri procedura se prenose putem stoga, međutim najčešće se kao parametri prenose samo adrese pokazivača na podatkovne strukture i vrlo je vjerojatno da same podatkovne strukture u trenutku pristupa neće biti u priručnoj memoriji.

Treća razina klasifikacije je prema tipu operanda u operaciji: skalarne varijable (engl. *variables*) i polja (engl. *arrays*) od jedne ili više dimenzija. Operacije koje sadrže operande zadane kao pokazivače (engl. *pointers*) se tretiraju kao skalarne varijable ukoliko se njihova vrijednost računa na temelju vrijednosti jedne varijable ili kao polja ukoliko se računaju na temelju vrijednosti više varijabli.

U skupovima *INT* i *FP* definirane su sljedeće elementarne operacije:

- *ADD* - zbrajanje koje podrazumijeva i oduzimanje,
- *MUL* - množenje,

- *DIV* - dijeljenje.

U skupu *LOG* definirane su sljedeće elementarne logičke operacije:

- *LOG* - uključuje logičke operacije *i*, *ili*, *isključivi ili* i *negacija*,
- *SHIFT* - uključuje logički ili aritmetički posmak te rotaciju.

Skup memorijskih operacija ima definirane elementarne operacije:

- *ASSIGN* - dodjeljivanje vrijednosti memorijskoj lokaciji,
- *BLOCK* - prijenos bloka podataka veličine 1000 u jedinstvenoj transakciji; može imati samo argumente tipa polje,
- *PROC* - poziv procedure. s jednim argumentom i povratnom vrijednosti; argument može biti varijabla ili polje, deklariran lokalno ili kao parametar procedure.

Sve elementarne operacije navedene su u tablici 3.1 koja je organizirana prema ranije navedenim razinama klasifikacije. Imena elementarnih operacija navedena u tablici koriste se u nastavku rada za identifikaciju elementarnih operacija u izvornom kôdu.

Tablica 3.1: Klasifikacijska shema elementarnih operacija

		Cjelobrojne operacije	Operacije s pomičnim zarezom	Logičke operacije	Memorijske operacije
lokalni operandi	varijable	INT_loc_var ADD INT_loc_var MUL INT_loc_var DIV	FP_loc_var ADD FP_loc_var MUL FP_loc_var DIV	LOG_loc_var LOG LOG_loc_var SHIFT	MEM_loc_var ASSIGN MEM_loc_var PROC
	polja	INT_loc_arr ADD INT_loc_arr MUL INT_loc_arr DIV	FP_loc_arr ADD FP_loc_arr MUL FP_loc_arr DIV	LOG_loc_arr LOG LOG_loc_arr SHIFT	MEM_loc_arr ASSIGN MEM_loc_arr BLOCK MEM_loc_arr PROC
globalni operandi	varijable	INT_glob_var ADD INT_glob_var MUL INT_glob_var DIV	FP_glob_var ADD FP_glob_var MUL FP_glob_var DIV	LOG_glob_var LOG LOG_glob_var SHIFT	MEM_glob_var ASSIGN
	polja	INT_glob_arr ADD INT_glob_arr MUL INT_glob_arr DIV	FP_glob_arr ADD FP_glob_arr MUL FP_glob_arr DIV	LOG_glob_arr LOG LOG_glob_arr SHIFT	MEM_glob_arr ASSIGN MEM_glob_arr BLOCK
operandi parametri procedura	varijable	INT_par_var ADD INT_par_var MUL INT_par_var DIV	FP_par_var ADD FP_par_var MUL FP_par_var DIV	LOG_par_var LOG LOG_par_var SHIFT	MEM_par_var ASSIGN MEM_par_var PROC
	polja	INT_par_arr ADD INT_par_arr MUL INT_par_arr DIV	FP_par_arr ADD FP_par_arr MUL FP_par_arr DIV	LOG_par_arr LOG LOG_par_arr SHIFT	MEM_par_arr ASSIGN MEM_par_arr BLOCK MEM_par_arr PROC

U izvornom kôdu na slici 3.1 ilustrirano je nekoliko primjera elementarnih operacija. Svaka operacija označena je kraticom iz tablice 3.1.

```
1  int g_var1 = 56;
2  float g_var2 = 5.6;
3
4  void function (int *a, int *d, float *f, float *g)
5  {
6      int x, y, z, b[100], c[100], i;
7      float e, h[100];
8
9      ...
10     z = y-x;                  // INT_loc_var ADD
11     g_var1 += i;              // INT_glob_var ADD
12     b[i] = a[i] + d[i];       // INT_par_arr ADD
13
14     e= y/x;                   // FP_loc_var DIV
15     g_var2 = g_var1 * y;      // FP_glob_var MUL
16     g[i] = f[i] * i;          // FP_par_arr MUL
17
18
19     for(i=1;i<100;i++){
20         d[i] = b[i];          // MEM_par_arr ASSIGN
21         c[i] = c[i] & b[i]; //  LOG_loc_arr LOG
22         d[i] = d[i-1]>>1;   //  LOG_par_arr SHIFT
23     }
24
25     ...
26 }
```

Slika 3.1: Primjer identifikacije elementarnih operacija u izvornom kôdu

3.2.2 Primjena klasifikacije na tipičnim konstruktima u kôdu aplikacija

Prezentirana klasifikacija operacija iz C izvornog kôda u elementarne operacije je načelno vrlo pojednostavljen pristup. U njemu nisu definirani specijalni slučajevi poput operacija s različitim tipovima operanada, prisutnost više operacija u jednom izrazu i sl. kakvi su redovito prisutni u kôdu raznih aplikacija. Kako bi se ispitalo kolika je točnost procjene trajanja izvođenja prema predloženoj klasifikacijskoj shemi za takve slučajeve, provedena je analiza na dva RISC procesora *ARM Cortex-A9* i *Microblaze* na platformi *Xilinx Zynq ZC706*. U tu svrhu izrađena je skupina ispitnih slučajeva koji predstavljaju tipične konstrukte kakvi se mogu pronaći u kôdu stvarnih aplikacija.

Prvi korak analize bilo je mjerenje vremena izvođenja svake elementarne operacije navedene u tablici 3.1 na obje arhitekture. Pri tome se svaka operacija izvela u *for*-petlji tisuću puta kako bi se ublažili vremenski učinci uspostavljanja vremenskog brojila i stvorio kontekst u kojem će se bolje imitirati učinci optimizacije i sklopovskih značajki kao što su priručna memorija i cjevovod. Primjer izvornog kôda mjernog programa za operaciju INT_loc_var ADD dan je na

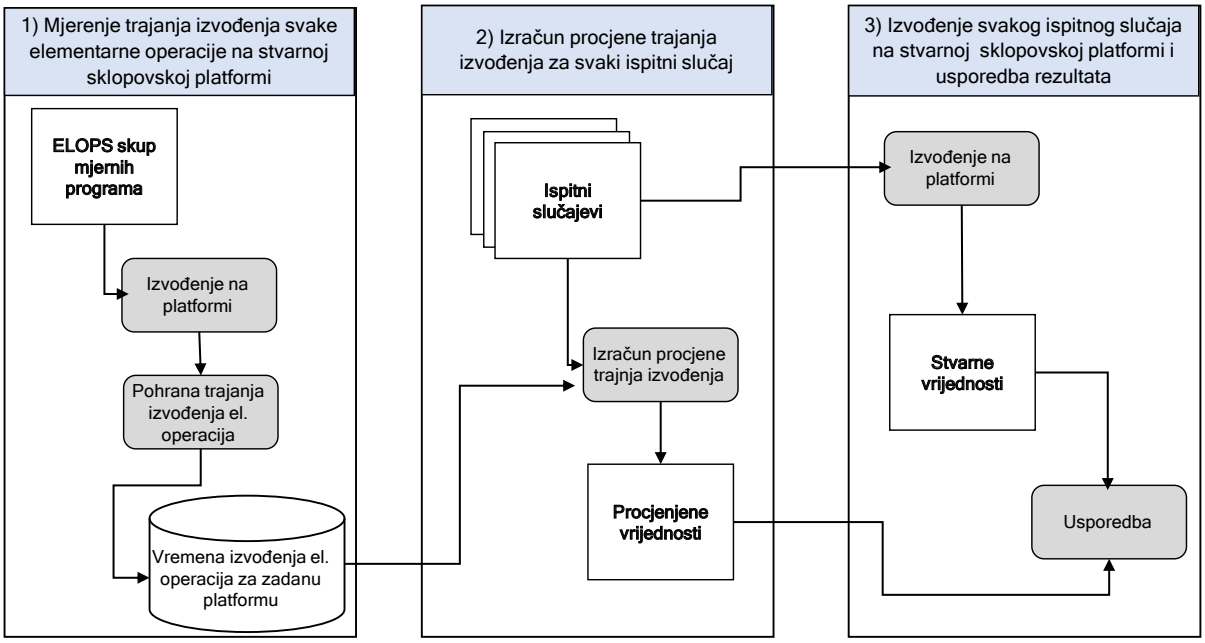
Slici 3.2. Sve ostale operacije su mjerene na isti način.

```
1     int a= ...;
2     int b= ...;
3     int c= ...;
4
5     char op_name[100];
6
7     timer_setup();
8
9     strcpy(op_name, "INT_loc_var - ADD");
10
11    t_start = get_timer_value();
12
13    for (a=0; a<1000; a++){
14        c=c+a;
15    }
16
17    t_end = get_timer_value();
18    duration = (t_end - t_start)/(double)a;
19    exec_time_s = duration/TICKS_PER_SEC;
20
21    store_exec_time(mem_loc, exec_time_s, op_name);
22
23    temp_res = c;
```

Slika 3.2: Kôd mjernog programa za operaciju INT_loc_var ADD

U kôdu mjernog programa moguće je na samom kraju primijetiti globalnu varijablu *temp_res* u kojoj se pohranjuje krajnji rezultat izračuna iz *for*-petlje. To je uvedeno zato što bi pri optimizaciji razine O1 i O2 prevoditelj mogao zaključiti da se vrijednost varijable *c* nigdje dalje ne koristi i potpuno izbaciti cijelu *for*-petlju [39]. Za mjerne programe u kojima su operandi polja, primijenjen je isti postupak s time što se rezultat pohranjuje u globalno polje, za sve elemente polja.

U drugom koraku analize najprije su za svaki ispitni slučaj identificirane elementarne operacije prisutne u tom slučaju. Zatim je za svaki ispitni slučaj, pomoću ranije dobivenih vremena izvođenja elementarnih operacija, izračunata procjena trajanja izvođenja cijelog slučaja. U trećem koraku su svi ispitni slučajevi u potpunosti izvedeni na ispitnim arhitekturama te su stvarno izmjerene vrijednosti trajanja izvođenja uspoređene s ranije procijenjenim vrijednostima. Tijek analize je ilustriran na Slici 3.3.



Slika 3.3: Tijek analize ispitnih slučajeva

Niz operacija

Niz (engl. *sequence*) elementarnih operacija iste vrste je vrlo često prisutan u bilo kojem tipu aplikacije. Moguće je imati više operacija iste vrste unutar jednog izraza (engl. *statement*) ili niz izraza sa po jednom operacijom iste vrste. Isječak izvornog kôda na Slici 3.4 ilustrira nekoliko primjera takvih slučajeva:

1. *SL1* - izraz u kojem je 5 slijednih cjelobrojnih operacija zbrajanja lokalnih varijabli *INT_loc_var ADD*,
2. *SL2* - *for*-petlja koja se izvodi 1000 puta i sadrži pet slijednih izraza od kojih svaki sadrži jednu cjelobrojnu operaciju zbrajanja lokalnih polja *INT_loc_arr ADD*,
3. *SL3* - izraz u kojem je 5 slijednih operacija množenja lokalnih varijabli s pomičnim zarezom *FP_loc_var MUL*,
4. *SL4* - *for*-petlja koja se izvodi 1000 puta i sadrži pet slijednih izraza od kojih svaki sadrži jednu operaciju množenja lokalnih polja s pomičnim zarezom *FP_loc_arr MUL*

Svaki slučaj SL1 - SL4 postaje jedan ispitni slučaj. Procjena trajanja izvođenja za svaki ispitni slučaj se računa tako da se zbroje trajanja svih elementarnih operacija prisutnih u tom ispitnim slučaju. Na primjer, za slučajeve SL1 i SL3 procijenjeno trajanje izvođenja će biti jednako zbroju trajanja izvođenja 5 operacija *INT_loc_var ADD*, odnosno *FP_loc_var MUL*. Za slučaj SL2 procijenjeno trajanje izvođenja će biti jednako zbroju trajanja izvođenja 5 operacija

```
1  void function ()
2  {
3      int a, b, c, d, e, f, g;
4      int x[1000], y[1000], z[1000], m[1000], n[1000];
5      float fa, fb, fc, fd, fe, ff, fg;
6      float f_x[1000], f_y[1000], f_z[1000], f_m[1000], f_n[1000];
7
8      c = a + b + d + e + f + g;          // SL1: 5 operacija INT_loc_var ADD
9                                          //      u 1 izrazu
10
11
12     for (i=0; i<1000; i++) {
13       z[i] = x[i] + y[i];               // SL2: 5 slijednih izraza s jednom
14       y[i] = x[i] + z[i];               //      INT_loc_arr ADD operacijom
15       m[i] = y[i] + z[i];
16       n[i] = m[i] + x[i];
17       x[i] = m[i] + n[i];
18     }
19
20     fc = fa * fb * fd * fe * ff * fg;   // SL3: 5 operacija FP_loc_var MUL
21                                         //      u 1 izrazu
22
23     for (i=0; i<1000; i++) {
24       f_z[i] = f_x[i] * f_y[i];         // SL4: 5 slijednih izraza s jednom
25       f_y[i] = f_x[i] * f_z[i];         //      FP_loc_arr MUL operacijom
26       f_m[i] = f_y[i] * f_z[i];
27       f_n[i] = f_m[i] * f_x[i];
28       f_x[i] = f_m[i] * f_n[i];
29     }
30 }
```

Slika 3.4: Primjeri nizova elementarnih operacija

INT_loc_arr ADD, odnosno FP_loc_arr MUL, pomnoženom s 1000 tj. brojem ponavljanja *for*-petlje. Potrebno je istaknuti da se u ovom pristupu *for*-petlje smatraju kao implicitni dio operacija s poljima kao tipom operanada. Ovo je direktna posljedica strukture mjernih programa u kojima se svaka elementarna operacija izvodi u *for*-petlji tisuću puta. Time je sav vremenski višak (engl. *overhead*) koji dodaje petlja već uključen u vrijednost koja se dobije izvođenjem ispitnog skupa.

Stvarne vrijednosti trajanja izvođenja za ove ispitne slučajeve dobivene su tako da je svaki slučaj izveden zasebno na platformama *ARM* i *Microblaze*. Mjerenje je za sve slučajeve napravljeno pomoću sistemskog brojača na način prikazan na slici 3.2. Za slučajeve SL1 i SL3 je linija kôda br. 14 sa slike 3.2 zamijenjena kôdom slučajeva SL1 odnosno SL3, a za slučajeve SL2 i SL4 je cijela *for*-petlja iz kôda sa slike 3.2 zamijenjena *for*-petljom iz SL2 odnosno SL4.

Procijenjene i stvarno izmjerene vrijednosti prikazane su u tablici 3.2. Pogreška procjene izračunata je prema formuli:

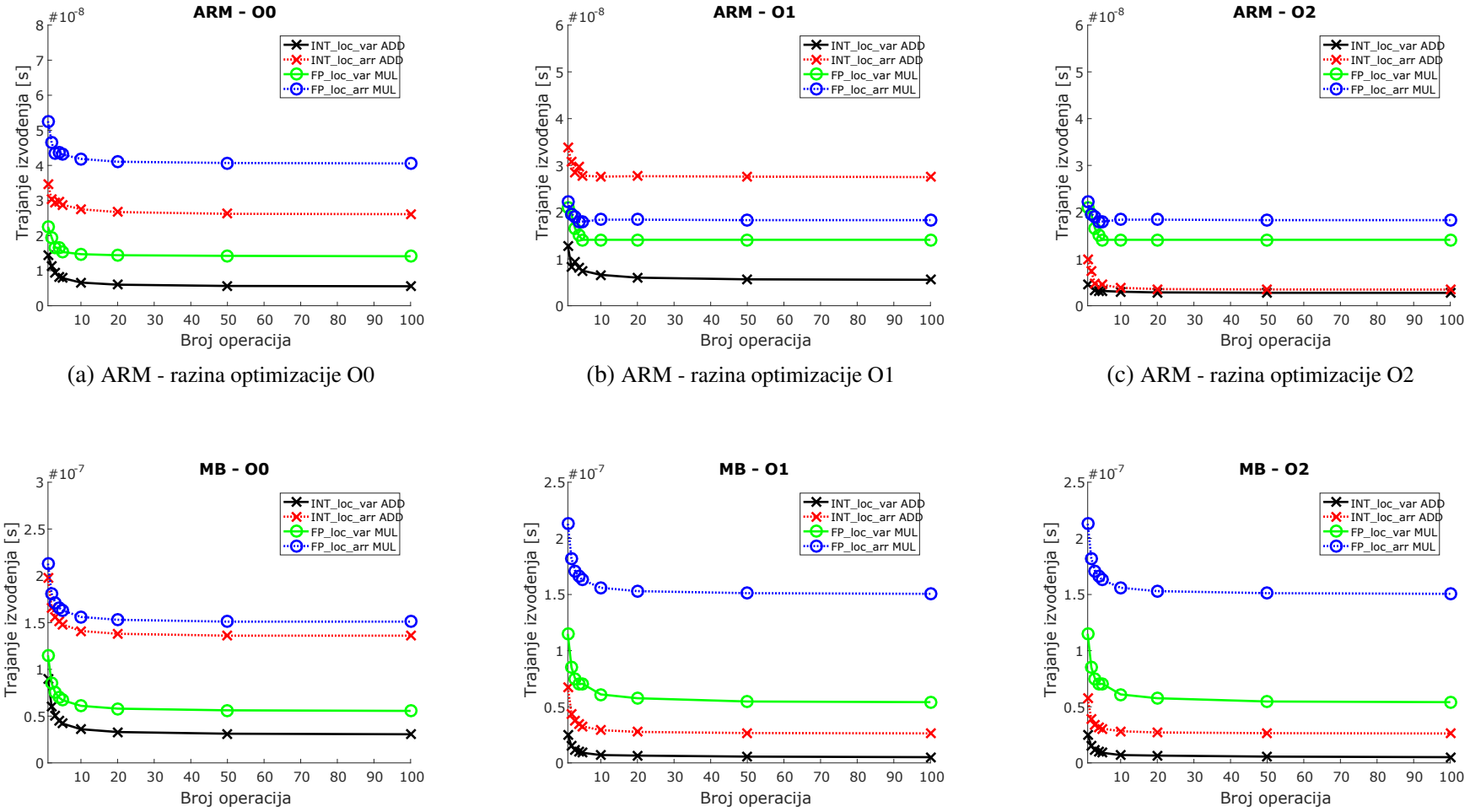
$$Pogreska = \frac{Procjenjeno\_vrijeme - Stvarno\_vrijeme}{Stvarno\_vrijeme} * 100\% \tag{3.1}$$

Tablica 3.2: Vremena izvođenja za operacije sa slike 3.4

		SL1	SL2	SL3	SL4
ARM	Procjenjeno vrijeme [s]	7.13E-08	2.57E-04	1.13E-07	2.57E-04
	Stvarno vrijeme [s]	2.14E-08	2.16E-04	4.20E-08	2.15E-04
	Pogreška [%]	233.18	18.98	169.05	19.53
Microblaze	Procjenjeno vrijeme [s]	4.50E-07	9.89E-04	5.75E-07	1.06E-03
	Stvarno vrijeme [s]	1.70E-07	7.38E-04	2.55E-07	8.13E-04
	Pogreška [%]	164.71	34.01	125.49	30.38

U svim slučajevima pogreška je veća od 20%, a u slučajevima SL1 i SL3 prelazi 100%. Uslijed ovako velike pogreške procjene, provedena je dodatna analiza kako bi se detaljnije ispitao utjecaj broja operacija u nizu na trajanje izvođenja po operaciji. Na procesorima ARM i Microblaze je za sve vrste elementarnih operacija i za razine optimizacije prevoditelja od O0 do O2 izmjereno trajanje izvođenja nizova izraza duljine: 2, 3, 4, 5, 10, 20, 50 i 100 s po jednom operacijom u izrazu.

Rezultati analize za cjelobrojno zbrajanje lokalnih varijabli *INT_loc_var ADD*, cjelobrojno zbrajanje lokalnih polja *INT_loc_arr ADD* te množenje lokalnih varijabli s pomičnim zarezom *FP_loc_var MUL* i množenje lokalnih polja s pomičnim zarezom *FP_loc_arr MUL* prikazani su na Slici 3.5.



(a) ARM - razina optimizacije O0

(b) ARM - razina optimizacije O1

(c) ARM - razina optimizacije O2

(d) Microblaze - razina optimizacije O0

(e) Microblaze - razina optimizacije O1

(f) Microblaze - razina optimizacije O2

Slika 3.5: Trajanje izvođenja po jednoj operaciji u ovisnosti o ukupnom broju operacija u nizu

Uočljivo je da vrijeme izvođenja po elementarnoj operaciji eksponencijalno pada s porastom ukupnog broja operacija u nizu. Takvo ponašanje je uočeno za sve tipove operacija, no zbog preglednosti i sažetosti prikazani su samo navedeni primjeri. Razlog ovakvog ponašanja se prvenstveno može objasniti razdjeljivanjem vremenskog troška koji unosi sama *for*-petlja (engl. *overhead*) kroz operacije ispitivanja uvjeta i uvećanja inkrementa petlje s jedne na više elementarnih operacija. Budući da operacije u nizu najčešće rade nad istim podacima smanjit će se i utjecaj promašaja priručne memorije. Također, kod slijednog izvođenja operacija nema prekida u tijeku izvođenja, tj. grananja, pa će biti izbjegnuti upravljački hazardi u cjevovodu procesora. Iz navedenog slijedi da je pri izračunu procjene trajanja izvođenja nužno prepoznati i uzeti u obzir moguće postojanje nizova operacija iste vrste u izvornom kôdu, kao i razlikovati da li se radi o nizu operacija unutar jednog izraza ili o nizu izraza s operacijama iste vrste.

Analizom rezultata sa slike 3.5 vidljivo je da za nizove duljine između 2 i 5, razlika u trajanju izvođenja između dva niza čija se duljina razlikuje za 1 (npr. 2 i 3, 3 i 4 itd.) može biti od 10 do 50%. Za nizove duljina od 5 do 10 razlika u trajanju izvođenja po operaciji je manje od 15%. A za još dulje nizove: 20, 50 i 100 razlika u trajanju izvođenja po operaciji je ispod 5%. Iz toga slijedi da je za nizove s više od 10 operacija dovoljno uzeti izmjereno trajanje po operaciji za niz duljine 10 te ga proporcionalno skalirati i pogreška će biti ispod 10%. Također se na temelju razlika u trajanju izvođenja po operaciji za nizove različitih duljina može zaključiti da nije potrebno mjeriti trajanje izvođenja za svaku moguću duljinu niza od 2 do 10 nego je dovoljno uzeti vrijednosti za jednu, dvije, tri, pet i deset operacija u nizu i na temelju toga aproksimirati za sve ostale duljine, a pogreška će ostati na razini ispod 10%.

Operacije s miješanim tipom i porijeklom operanada

Razumno je za pretpostaviti da je u kôdu neke aplikacije moguće redovito susresti operacije s miješanim tipom operanada. Isječak izvornog kôda na Slici 3.6 prikazuje nekoliko takvih slučajeva.

Klasifikacijska shema u ovim slučajevima ne daje nikakve smjernice kako odrediti kojoj vrsti elementarne operacije pripada pojedina operacija iz kôda. Stoga se dodatno uvodi novi element u klasifikaciju - *prioritet porijekla* pomoću kojeg će se odrediti vrsta elementarne operacije u slučaju operanada različitog porijekla. Prioriteti su definirani na temelju razlika u načinu na koji prevoditelj implementira (smješta u memorijskom prostoru) različite tipove operanada. Redoslijed je zadan od najvišeg prema najnižem prioritetu, a vrsta elementarne operacije se

```
1  int g_a, g_c;
2  int gf[1000], gh[1000];
3  void function ()
4  {
5      int i;
6      int x[1000];
7
8      g_c=g_a+i;                    // SL1: INT_glob_var ADD
9
10     for (i=0; i<1000; i++)
11       gf_h[i] = gf[i] + x[i];     // SL2: INT_glob_arr ADD
12
13     for (i=0; i<1000; i++)
14       gf_h[i] = gf[i] + i;        // SL3: INT_glob_arr ADD
15
16     for (i=0; i<1000; i++)
17       gf_h[i] = x[i] + i;         // SL4: INT_glob_arr ADD
18 }
```

Slika 3.6: Primjer operacija s miješanim tipom i porijeklom operanada

određuje prema operandu najvišeg prioriteta:

1. polje kao parametar procedure
2. globalno polje
3. lokalno polje
4. varijabla kao parametar procedure
5. globalna varijabla
6. lokalna varijabla

Za primjer sa slike 3.6, u svakom od slučajeva svaka će operacija biti klasificirana kao operacija s globalnim operandima, iako operacije u slučajevima SL1, SL3 i SL4 sadrže lokalne varijable, a operacije u slučajevima SL2 i SL4 sadrže lokalna polja.

Analiza točnosti procjene trajanja izvođenja za primjer sa slike 3.6 provedena je na način da svaki slučaj SL1 - SL4 postaje jedan ispitni slučaj. Pri procjeni trajanja izvođenja primjenjen je isti postupak kao i pri analizi nizova operacija u prethodnoj sekciji. Mjerenje stvarne vrijednosti trajanja izvođenja za ove ispitne slučajeve također je provedeno na isti način kao i u prethodnoj sekciji.

Usporedba procijenjenih i stvarno izmjerenih vrijednosti trajanja izvođenja prikazana je u tablici 3.3. Za slučajeve SL2 i SL3 pogreška procjene je veća od 20%. Za slučajeve SL1 i SL4 pogreška u procjeni je ispod 2%, no to je iz razloga što u slučaju kad su u operaciji prisutna lokalna polja metoda daje premalu procijenjenu vrijednost (kao u slučaju SL2), a kada su u operaciji prisutne lokalne varijable daje preveliku procijenjenu vrijednost (kao u slučaju SL3). Kako su u ovom slučaju prisutne obje vrste operanada, pogreške u procjeni se međusobno

poništavaju.

Tablica 3.3: Vrijeme izvođenja operacija sa slike 3.6

		SL1	SL2	SL3	SL4
ARM	Procijenjeno vrijeme [s]	2.89E-08	3.48E-05	3.48E-05	3.48E-05
	Stvarno vrijeme [s]	2.85E-08	4.73E-05	2.86E-05	3.53E-05
	Pogreška [%]	1.33	-26.49	21.85	-1.36
Microblaze	Procijenjeno vrijeme [s]	1.05E-07	1.83E-04	1.83E-04	1.83E-04
	Stvarno vrijeme [s]	1.00E-07	1.88E-04	1.53E-04	1.58E-04
	Pogreška [%]	5.00	-2.66	19.61	15.82

U cjelini, ovakvi rezultati ukazuju na to da je nužna modifikaciju pristupa na način da se kao i ranije pretpostavi vrsta elementarne operacije prema prioritetu operanada, ali da se uvede i dodatni atribut u kojem će se naznačiti postojanje operanada drukčijeg tipa. Stoga se uvodi atribut pod nazivom *mod* u kojem će se naznačiti postojanje operanada drukčijeg tipa. Vrijednost ovog atributa treba omogućiti razlikovanje sljedećih slučajeva:

- prisutnost varijabli u operaciji s poljima istog porijekla,
- prisutnost globalnih varijabli u operacijama s parametrima procedure,
- prisutnost globalnih polja u operacijama s poljima koja su parametri procedure,
- prisutnost lokalnih varijabli u operacijama s globalnim operandima ili parametrima procedure,
- prisutnost lokalnih polja u operacijama s globalnim ili poljima parametrima procedure,
- prisutnost konstanti.

Popis mogućih vrijednosti atributa *mod* je dan u tablici 3.5 u stupcu *Vrijednost*.

Indeks u operandima tipa polje

Operandi tipa polje mogu imati jednodimenzionalni ili višedimenzionalni indeks, tj. mogu se računati na temelju vrijednosti jedne ili više varijabli. Isto je primjenjivo i na *struct* tip u C izvornom kôdu koji može imati više od jedne razine tj. sadržavati polja od jedne ili više dimenzija. Isječak izvornog kôda na Slici 3.7 ilustrira nekoliko primjera. Slučaj SL1 je primjer operacije dodjeljivanja vrijednosti memorijskoj lokaciji za trodimenzionalno polje. SL2 je sličan primjer sa tipom *struct* koji u sebi sadrži dvodimenzionalno polje. Prema predloženom modelu obje

operacije se klasificiraju kao operacije s poljima. Slučajevi SL3 do SL7 su primjeri operacija s poljima s višedimenzionalnim indeksima koji se računaju na temelju vrijednosti nekoliko varijabli.

```
1  int gf_f[1000], gf_g[1000], gf_h[1000];
2  void function (struct_i *Y, struct_i *X, int *h, int *f)
3  {
4      ...
5
6      for (i=0; i<10; i++)
7          for (j=0; j<10; j++)
8              for (m=0; m<10; m++)
9                  h[i][j][m] = f[i][j][m];                // SL1: MEM_par_arr ASSIGN
10
11     for (i=0; i<10; i++)
12         for (j=0; j<10; j++)
13             for (m=0; m<10; m++)
14                 Y[i].field[j][m] = X[i].field[j][m];    // SL2: MEM_par_arr ASSIGN
15
16     for (i=0; i<1000; i++)
17         gf_f[i] = gf_f[(i*5)%1000] + gf_h[i];           // SL3: INT_glob_arr ADD
18     for (i=0; i<1000; i++)
19         gf_g[i] = gf_f[(i*7+2090)%1000] + gf_h[i];      // SL4: INT_glob_arr ADD
20     for (i=0; i<1000; i++)
21         gf_g[i] = gf_f[(i*7+x*3)%1000] + gf_h[i];       // SL5: INT_glob_arr ADD
22     for (i=0; i<1000; i++)
23         gf_g[i] = gf_f[(i*7+x*3+7)%1000] + gf_h[i];     // SL6: INT_glob_arr ADD
24     for (i=0; i<1000; i++)
25         gf_g[i] = gf_f[(i*7+x*3+y*5)%1000] + gf_h[i];   // SL7: INT_glob_arr ADD
26
27     ...
28 }
```

Slika 3.7: Primjeri različitih vrsta indeksa u operandima tipa polje

Za svaki slučaj SL1 do SL7 napravljen je poseban ispitni slučaj i proveden na isti način kao što je opisano u prethodnim sekcijama. U tablici 3.4 navedena su procijenjena i stvarna vremena izvođenja. Pogreška procjene je između 50 i 100% što implicira da se predložena klasifikacijska shema mora dodatno proširiti kako bi se dobili točniji rezultati. Gotovo identična razina pogreške je uočena i u slučajevima SL1 s 3-dimenzionalnim poljem i SL2 sa *struct*-om koji u sebi sadrži 2-dimenzionalno polje (što ga čini 3-dimenzionalnom strukturom). Za slučajeve SL3 do SL7 može se primijetiti da su rezultati značajno podcijenjeni te da razina pogreške raste kako se povećava broj operacija koje se moraju izvršiti da bi se izračunala vrijednost indeksa polja.

Iz rezultata se može zaključiti da je nužno proširiti temeljni pristup te pri procjeni uzeti u obzir broj dimenzija polja i strukturu indeksa. Budući da se stvarno izmjerena vremena za obična polja i *struct*-ove iste veličine ne razlikuju, pri klasifikaciji će se strukture i dalje tretirati kao

Tablica 3.4: Vrijeme izvođenja za operacije sa slike 3.7

		SL1	SL2	SL3	SL4	SL5	SL6	SL7
ARM	Procijenjeno vrijeme [s]	2.55E-05	2.55E-05	3.48E-05	3.48E-05	3.48E-05	3.48E-05	3.48E-05
	Stvarno vrijeme [s]	6.94E-05	6.26E-05	6.25E-05	6.15E-05	6.80E-05	6.86E-05	7.88E-05
	Pogreška [%]	-63.21	-59.30	-44.32	-43.44	-48.81	-49.28	-55.81
Microblaze	Procijenjeno vrijeme [s]	1.60E-04	1.60E-04	1.83E-04	1.83E-04	1.83E-04	1.83E-04	1.83E-04
	Stvarno vrijeme [s]	2.42E-04	2.52E-04	2.58E-04	2.73E-04	2.93E-04	2.98E-04	3.28E-04
	Pogreška [%]	-33.88	-36.51	-29.07	-32.97	-37.54	-38.59	-44.21

polja. Vezano uz vrstu i dimenziju indeksa polja, kao rješenje se predlaže dodavanje posebnih atributa osnovnoj operaciji pomoću kojih će se naznačiti:

1. vrstu indeksa - *type*
   - može biti jednostavan - *simple*, složen - *complex* ili konstanta - *const*
2. dimenziju polja - *dim*
3. broj zbrajanja koje je potrebno izvršiti da bi se izračunao indeks polja za složeni tip polja - *add_nr*
4. broj množenja koje je potrebno izvršiti da bi se izračunao indeks polja za složeni tip polja - *mul_nr*.

Navedeni atributi sistematizirani su u tablici 3.5 u dijelu *modifikatori indeksa*.

3.2.3 Atributi klasifikacijske sheme elementarnih operacija

Na temelju ranije izloženih opažanja dopunjuje se klasifikacijska shema kako bi obuhvatila predložena rješenja. Svaka vrsta elementarnih operacija dobiva atribute pomoću kojih se opisuju dodatna svojstva. Prema tri posebna slučaja koja utječu na trajanje izvođenja elementarnih operacija definiraju se tri skupine atributa navedene u tablici 3.5 u stupcu *Grupa atributa*.

Postojanje niza operacija iste vrste u jednom izrazu naznačavat će se atributom *seq* koji može imati pozitivnu cjelobrojnu vrijednost. Atribut *mod* bit će prisutan u onim operacijama gdje su operandi miješanog tipa i porijekla. Vrijednost ovog atributa je lista koja sadrži jednu ili više vrijednosti navedenih u tablici 3.5 u stupcu *Vrijednosti*. Atributi kojima će se opisivati dimenzija i vrsta indeksa u operacijama s poljima nalaze se u grupi *Modifikatori indeksa*. To su ukupno četiri atributa: *type*, *dim*, *add_nr* i *mul_nr* koji detaljnije opisuju operacije s poljima na način kako je navedeno u poglavlju 3.2.1.

Tablica 3.5: Atributi elementarnih operacija

Grupa atributa	Ime atributa	Vrijednosti
niz operacija	*seq*	pozitivni cijeli broj
modifikator operacije	*mod*	"var" - prisutna je najmanje jedna varijabla istog porijekla "glob_var" - prisutna je najmanje jedna globalna varijabla "glob_arr" - prisutno je najmanje jedno globalno polje "loc_var" - prisutna je najmanje jedna lokalna varijabla "loc_arr" - prisutno je najmanje jedno lokalno polje "const" - prisutna je najmanje jedna konstanta
modifikator indeksa	*type*	"simple" - index je zadan kao jedna varijabla "complex" - indeks se računa pomoću dvije ili više varijabli "const" - indeks je konstanta
	dim	pozitivni cijeli broj - dimenzija indeksa polja
	add_nr	pozitivni cijeli broj - broj operacija zbrajanja potrebnih za izračun indeksa
	mul_nr	pozitivni cijeli broj - broj operacija množenja potrebnih za izračun indeksa

3.3 Postupak procjene trajanja izvođenja aplikacije

U osnovi metode ELOPS-EM je izrada profila aplikacije i platforme korištenjem prethodno opisanog knocepta elementarnih operacija. Na temelju tih profila, poseban algoritam računa procjenu trajanja izvođenja.

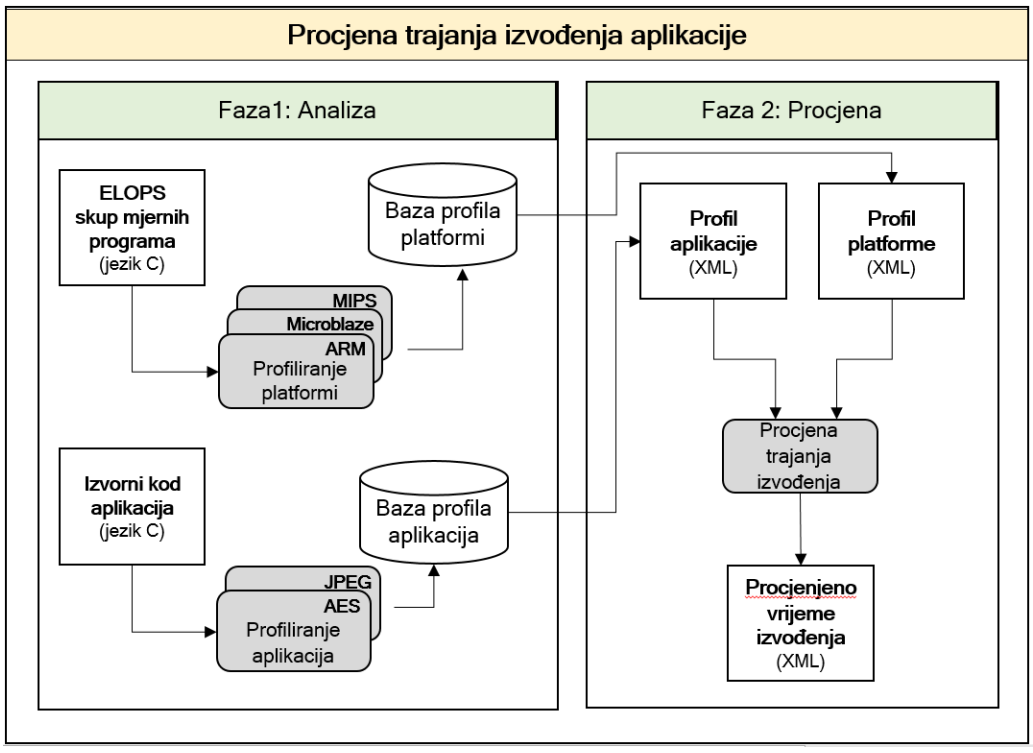
Postupak procjene trajanja izvođenja prema metodi ELOPS-EM sastoji se od faze analize i faze procjene, kao što je prikazano na slici 3.8. Prvi korak faze analize je profiliranje platforme. U tom koraku se posebno izrađeni skup mjernih programa *ELOPS*, detaljno opisan u poglavlju 3.3.1, prevodi i pokreće zasebno za svaku konfiguraciju platforme i za svaku razinu optimizacije prevedenog kôda. Na temelju rezultata izvođenja mjernog programa izgrađuje se profil platforme koji sadrži vremena izvođenja svih vrsta elementarnih operacija. Drugi korak u fazi analize je profiliranje aplikacije koje se izvodi samo jednom i to na izvornom C kôdu bez prevođenja. Pri tome se koriste transformacije kôda u konstrukte kao što su apstraktno sintaksno stablo AST (engl. *abstract syntax tree*) i graf tijeka upravljanja i podataka CDFG (engl. *control and data flow graph*), a krajnji rezultat je profil aplikacije u obliku liste elementarnih operacija strukturiranih u petlje, grane i nizove. Svi dobiveni profili aplikacija i platforma se

trajno pohranjuju u bazu podataka kako bi se u budućnosti mogli ponovno koristiti. To znači da ako se želi ista aplikacija izvesti na drugoj platformi nije potrebno ponavljati profiliranje aplikacije u fazi analize. Isto vrijedi i za platformu: jednom profilirana platforma se ne mora ponovno profilirati ako se koristi za neku drugu aplikaciju.

U fazi procjene, najprije se dohvaćaju profili aplikacije i platforme iz baze podataka. Zatim se na temelju ta dva profila računa procjena trajanja izvođenja prema algoritmu opisanom potpoglavlju 3.3.3.

3.3.1 Profiliranje platforme

Profiliranje platforme počinje izvođenjem posebno izrađenog skupa mjernih programa na svakoj konfiguraciji platforme koju se razmatra za konačno rješenje. Pojam *konfiguracija* u ovom kontekstu predstavlja vrstu procesora i memorije u kojoj taj procesor pohranjuje podatke i instrukcije. Vrsta procesora je definirana tipom arhitekture (npr. RISC) te parametrima: radni takt, broj stupnjeva cjevovoda, veličina priručne memorije i prisutnost dodatnog sklopovlja poput sklopovskog množila, posmačnog registra itd. Za memoriju je definiran tip (npr. DDR3 SDRAM), radna frekvencija, kapacitet te broj i vrsta priključaka.



Slika 3.8: Tijek procjene trajanja izvođenja aplikacije

Skup mjernih programa *ELOPS* izrađen je na temelju klasifikacijske sheme elementarnih operacija iz tablice 3.1[a]. Pomoću njega je moguće izmjeriti trajanje svakog tipa operacije u klasifikacijskoj shemi, te učinke svakog atributa iz tablice 3.5 kao što su npr. slijedno izvođenje, različite vrste indeksa itd. Mjerni programi su sistematizirani u tablici 3.6 slijedeći klasifikacijsku shemu elementarnih operacija. Za svaku operaciju u temeljnim skupovima iz tablice 3.1 (npr. INTEGER ADD, LOGIC SHIFT itd.), definirane su tri glavne skupine mjernih programa prema tri moguća porijekla operanada: *lokalni*, *globalni* i *parametarski* - stupac *Porijeklo* u tablici 3.6. Svaka skupina sadrži po dvije podskupine: *varijable* i *polja*. Po ovim podskupinama su definirani *osnovni mjerni programi* - stupac *Osnovni mjerni programi*. Za podskupinu *varijable* definiran je jedan osnovni mjerni program. Za podskupinu *polja* su definirana četiri osnovna mjerna programa: tri za tip indeksa *simple*, po jedan za dimenzije indeksa od 1 do 3, i jedan za tip indeksa *complex*.

Svaki osnovni mjerni program ima pod-inačice u kojima se mijenja duljina niza operacija. Razlikuje se niz od više operacija iste vrste u jednom izrazu - *niz operacija*, i niz izraza s po jednom operacijom iste vrste - *niz izraza*. U trenutnoj implementaciji, postoje inačice osnovnih mjernih programa za sljedeće duljine nizova: 2, 3, 5 i 10. Nadalje, utjecaj atributa iz Tablice 3.5 mjeri se uvođenjem dodatnih pod-inačica osnovnih mjernih programa. Za svaku skupinu atributa iz tablice 3.6 u stupcu *Modifikatori* postoji zasebni mjerni program. Također za svaki takav mjerni program mjeri se trajanje izvođenja nizova izraza duljine 2, 3, 5 i 10. Sveukupno postoji oko 3000 mjernih programa.

[a] Skup mjernih programa je dostupan na adresi `https://gitlab.com/Frid/ELOPS`

Tablica 3.6: Sistematizacija mjernih programa ELOPS

Porijeklo	Osnovni mjerni programi		Modifikatori	Duljine nizova koje se mjere
lokalni operandi	varijable		const	jedna operacija niz operacija (duljine: 2,3,5,10) [b] niz izraza (duljine: 2,3,5,10)
	polja	*type*="simple", *dim*=1 *type*="simple", *dim*=2 *type*="simple", *dim*=3	var const	jedna operacija niz operacija (duljine: 2,3,5,10) ?? niz izraza (duljine: 2,3,5,10)
		type = "complex", *dim*=1	add_nr mul_nr	jedna operacija
globalni operandi	varijable		loc_var const	jedna operacija niz operacija (duljine: 2,3,5,10) ?? niz izraza (duljine: 2,3,5,10)
	polja	*type*="simple", *dim*=1 *type*="simple", *dim*=2 *type*="simple", *dim*=3	const var loc_var loc_arr	jedna operacija niz operacija (duljine: 2,3,5,10) ?? niz izraza (duljine: 2,3,5,10)
		type = "complex", *dim*=1	add_nr mul_nr	single
parametri procedure	varijable		glob_var loc_var const	jedna operacija niz operacija (duljine: 2,3,5,10) ?? niz izraza (duljine: 2,3,5,10)
	polja	*type*="simple", *dim*=1 *type*="simple", *dim*=2 *type*="simple", *dim*=3	const var loc_var loc_arr glob_var glob_arr	jedna operacija niz operacija (duljine: 2,3,5,10) ?? niz izraza (duljine: 2,3,5,10)
		type = "complex", *dim*=1	add_nr mul_nr	single

Svi mjerni programi su strukturirani na isti način - kako je prikazano u kôdu na slici 3.2. Poseban slučaj su memorijske operacije BLOCK i PROC. Operacija BLOCK se mjeri kao jedna transakcija bloka podataka veličine 1000 koristeći funkciju *memcpy* i može imati samo polja

kao operande. Operacija PROC se mjeri kao poziv funkcije s jednim argumentom i povratnom vrijednošću.

Kako bi se postigla visoka točnost pri procjeni trajanja izvođenja kôda koji u sebi sadrži optimizacije, izvorni kôd mjernih programa se prevodi i izvodi na ispitnoj platformi posebno za svaku razinu optimizacije. Na taj način iste optimizacije koje bi bile prisutne u kôdu aplikacije za npr. izvođenje operacija u petlji ili u nizu će biti prisutne u kôdu mjernog programa.

Profiliranje platforme mora se izvesti samo jednom za svaki konfiguracijski par procesor-memorija i ti se rezultati kasnije mogu koristiti bez ikakve dodatne dorade za bilo koju aplikaciju koja će se izvoditi na toj konfiguraciji.

Rezultati profiliranja se zapisuju kao specifikacija konfiguracije procesor-memorija platforme u XML strukturi. Vršni čvor je *platform* koji predstavlja platformu za koju je izvršeno profiliranje. Čvor *platform* ima po jedan podčvor *operation* za svaki osnovni mjerni program iz skupa *ELOPS* mjernih programa. Čvor *operation* obavezno ima atribut *name* u kojem je zapisan naziv elementarne operacije iz tablice 3.1, a može imati atribute *type* i *dim* ukoliko se radi o operacijama s poljima. Ti atributi označavaju vrstu i dimenziju indeksa polja respektivno. Djeca čvora *operation* su: *single* - samostalna elementarna operacija, *seqop* - niz operacija u jednom izrazu i *seqstat* - niz izraza. Za svaku duljinu niza koja se mjeri, stvara se novi *seqop* ili *seqstat* čvor, a njihov atribut *nr* označava duljinu niza. Unutar čvorova *single*, *seqop* i *seqstat* nalazi se čvor *base* te jedan ili više čvorova *mod*. Vrijednost čvora *base* je trajanje izvođenja osnovnog mjernog programa (bez učinka atributa). Vrijednost čvorova *mod* jest trajanje izvođenja operacije u kojoj je prisutan neki od mogućih modifikatora navedenih u stupcu *Modifikatori* u Tablici 3.6. Naziv modifikatora naveden je u atributu *type* čvora *mod*. Svi elementi profila platforme sistematizirani su u tablici 3.7.

Tablica 3.7: Elementi profila platforme

Element	Mogući pod-elementi	Atributi	
		Ime	*Opis*
platform	operation	name	*ime platforme*
		optlvl	*razina optimizacije*
operation	single, seqop, seqstat	name	*naziv elementarne operacije*
		type	*tip indeksa* [a]
		dim	*dimenzija indeksa polja*[a]
single	base, mod	-	-
seqop	base	nr	*broj ponavljanja operacije u izrazu*
seqstat	base, mod	nr	*broj ponavljanja izraza u nizu*
base	-	-	-
mod	-	type	*naziv modifikatora operacije ili indeksa*

[a] primjenjivo samo za operacije s poljima

Kratki izvadak iz profila procesora ARM Cortex-A9 za operaciju cjelobrojnog zbrajanja prikazan je na slici 3.9. U izvatku su prikazani isječci profila za operaciju INT_loc_var ADD i INT_loc_arr ADD. Cijeli profil za operaciju cjelobrojnog zbrajanja se nalazi u Dodatku A.

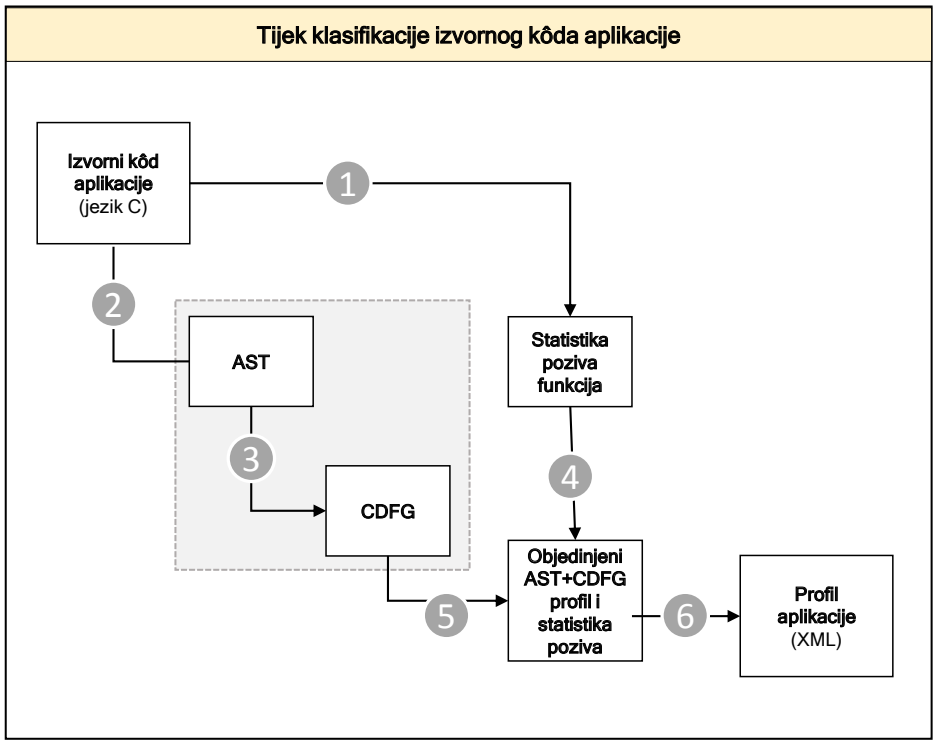
```
<platform name="ARM1" optlvl="0">
    <operation name="INT_loc_var ADD">
        <single>
            <base>1,351500e-08</base>
            <mod type="const">1,501470e-08</mod>
        </single>
        <seqop nr="2">
            <base>1,351500e-08</base>
        </seqop>

        ...

        <seqstat nr="2">
            <base> 2,251500e-08</base>
            <mod type="const">1,501710e-08</mod>
        </seqstat>

        ...
    </operation>
    <operation name="INT_loc_arr ADD" type="simple" dim="1">
        <single>
            <base>5,139000e-08</base>
            <mod type="var">3,900000e-08</mod>
            <mod type="const">3,753000e-08</mod>
        <seqop nr="2">
            <base>6,456000e-08</base>
        </seqop>
        <seqop nr="3">
            <base>7,983000e-08</base>
        </seqop>

    ...
```

Slika 3.9: Isječak iz primjera profila platforme procesora ARM Cortex-A9 za operaciju cjelobrojnog zbrajanja

3.3.2 Profiliranje aplikacije

Profiliranje aplikacije temelji se na metodi analize i profiliranja izvornog kôda opisanoj u [40], koja je modificirana kako bi bila kompatibilna s klasifikacijskom shemom elementarnih operacija [41]. Obrada izvornog kôda aplikacije počinje izgradnjom stabla poziva procedura (engl.*call tree*) - korak 1 na slici 3.10. Pomoću toga je moguće dobiti statističke informacije za profiliranje na razini grafa poziva procedure (engl. *call-graph*). Zatim slijed tijek transformacija koje radi prevoditelj kako bi se u danom izvornom kôdu identificirale elementarne operacije. Najprije se izvorni kôd parsira u apstraktno sintaksno stablo - AST (engl. *abstract syntax tree*) - korak 2. Tijekom rekurzivnog obilaska stabla, za identifikaciju elementarnih operacija se koriste informacije o podatkovnim strukturama, tipovima varijabli i argumentima procedura. AST se nadalje transformira u graf upravljačkog i podatkovnog toka - CDFG (engl. *control and data flow graph*) koristeći rekurzivni obilazak stabla i privremene varijable koje tvore tro-adresnu notaciju izraza izvornog kôda - korak 3. Tijekom tog procesa, ključni je element prepozna-

vanje entiteta koji usmjeravaju upravljački tok aplikacije. To su točke gdje se uniformni tijek instrukcija razbija u petlje ili grane s uvjetima. U sljedećem koraku se objedinjava statistika poziva procedura te profila dobivenih na temelju AST i CDFG transformacija za svaku proceduru zasebno - koraci 4 i 5. Izlaz iz procesa je profil aplikacije kao apstraktni model zapisan u XML strukturi u kojem se izvorni kod aplikacije transformira u višerazinsku strukturu koju čine elementarne operacije organizirane u petlje, grane i nizove - korak 6.



Slika 3.10: Tijek profiliranja aplikacije

U profilu se aplikacija sastoji od jedne ili više procedura koje direktno odgovaraju procedurama (funkcijama) u izvornom C kôdu. Procedure mogu sadržavati neograničeni broj procedura, petlji, uvjetnih grananja ili operacija. Petlja predstavlja *for*- ili *while*-petlju, a uvjetno grananje predstavlja *if-else* ili *switch-case* konstrukte. Petlje i grananja mogu imati neograničeni broj procedura, petlji, grana i operacija kao pod-elemente. Operacija predstavlja elementarnu operaciju iz tablice 3.1. Operacije mogu imati atribute u skladu s proširenjem klasifikacijske sheme predstavljenom u poglavlju 3.2.2. U XML strukturi vršni čvor je *application* koji predstavlja aplikaciju za koju se profilira. Čvor *application* može imati neograničeni broj čvorova *procedure* koji predstavljaju procedure (funkcije) u aplikaciji. Djeca čvora *procedure* mogu biti: *procedure* - procedure, *loop* - petlja, *branch* - uvjetno grananje i *operation* - operacija. Čvor

loop ima atribut *count* u kojemu je zapisan broj ponavljanja petlje. Čvor *branch* ima atribut *cond* u kojem je zapisan uvjet grananja i čvorove djecu: *truebody* i *falsebody* koje predstavljaju pojedine uvjetovane grane. Ti čvorovi imaju atribut *count* u kojemu je zapisan broj ponavljanja te grane. Vrijednost čvora *operation* je broj uzastopnih elementarnih operacija unutar jednog izraza (engl. *statement*). Svi elementi profila sistematizirani su u tablici 3.8.

Tablica 3.8: Elementi profila aplikacije

Element	Mogući pod-elementi	Atributi	
		Ime	*Opis*
application	procedure	name	ime aplikacije
procedure	procedure, loop, branch, operation	name	ime procedure
loop	procedure, loop, branch, operation	count	broj koliko puta se petlja izvede
branch	truebody, falsebody	cond	uvjet grananja
truebody	procedure, loop, branch, operation	count	broj koliko puta se element *true-body* izvede
falsebody	procedure, loop, branch, operation	count	broj koliko puta se element izvede
operation	-	class	klasa elementarne operacije (npr. INT_loc_var)
		type	operacija (e.g. add, mul...)
		mod	modifikator operacije (e.g. var, glob_var...)
		index_type	tip indeksa: *simple*, *complex* ili *const* [c]
		dim	broj dimenzija polja [a]
		add_nr	broj operacija zbrajanja potrebnih za izračun indeksa [d]
		mul_nr	broj operacija množenja potrebnih za izračun indeksa [??]

Za aplikacije koje imaju podatkovno-ovisno ponašanje, preciznost profiliranja ovisi o ulaznim podacima tijekom izvođenja. U takvim slučajevima broj iteracija petlji ili rezultat evaluacije uvjeta grananja nije moguće saznati bez simulacije. Budući da ove činjenice određuju broj mogućih putova kojima će se proći kroz kôd aplikacije, potrebno je provesti određeni broj simulacija kako bi se procijenila gornja i donja granica broja prolaska određenim putem. Simulacije je moguće provesti na različite načine, a u sklopu ovog istraživanja primijenjen je najčešće korišteni pristup - pokretanje instrumentiranog kôda na računalu domaćinu (engl. *host-compiled instrumentation*) [4, 7, 24]. Instrumentacija služi za određivanje statistike prolaska određenim dijelovima kôda, a izvođenjem kôda na računalu domaćinu (osobnom računalu opće namjene) eliminira se vrijeme i napor koje je potrebno uložiti za izgradnju simulatora ili prilagodbu kôda za izvođenje na stvarnoj MPSoC platformi.

Kratki izvadak iz profila aplikacije koja implementira algoritam *AES* [11], a u kojem prikazan profil za proceduru *KeyExpansion* dan je na slici 3.11. Profil cijele aplikacije nalazi se u Dodatku B.

```
<?xml version="1.0"?>
<app  name="AES">
    <procedure name="KeyExpansion">
        <loop count="4">
            <operation class="MEM_glob_var" type="assign" mod="const">1</
                operation>
            <operation class="MEM_glob_arr" type="assign" index_type="cmplx"
                 level="1" mul_nr="1">1</operation>
            <operation class="MEM_glob_arr" type="assign" index_type="cmplx"
                 level="1" mul_nr="1" add_nr="1">3</operation>
        </loop>
        <loop count="40">
            <loop count="4">
                <operation class="MEM_glob_var" type="assign" mod="const"
                    >1</operation>
                <operation class="MEM_glob_arr" type="assign" index_type="
                    cmplx" level="1" mul_nr="1" add_nr="2">3</operation>
            </loop>
            <branch cond="i%4==0">
                <truebody count="10">
                    <operation class="MEM_glob_var" type="assign" mod="const
                        ">1</operation>
                    <operation class="MEM_glob_arr" type="assign" mod="
                        loc_var" index_type="const" level="1">1</operation>
                    <operation class="MEM_glob_arr" type="assign" index_type
                        ="const" level="1">3</operation>
                    <operation class="MEM_glob_arr" type="assign" mod="
                        loc_var" index_type="const" level="1">1</operation>
                    <operation class="MEM_glob_arr" type="assign" index_type
                        ="const" level="1">4</operation>
                    <operation class="MEM_glob_arr" type="rfunc">4</
                        operation>
                </truebody>
                <falsebody count="1">
                    <branch cond="4&amp;gt;6 &amp;amp;&amp;amp; i%4==4">
                        <truebody  count="10">
                            <operation class="MEM_glob_var" type="assign"
                                mod="const">1</operation>
                            <operation class="MEM_glob_arr" type="assign"
                                index_type="const" level="1">4</operation>
                            <operation class="MEM_glob_arr" type="rfunc">4</
                                operation>
                        </truebody>
                        <falsebody>
                        </falsebody>
                    </branch>
                </falsebody>
            </branch>
            <operation class="MEM_glob_var" type="assign" mod="const">1</
                operation>
            <operation class="LOG_glob_arr" type="log" index_type="cmplx"
                level="1" mul_nr="1" add_nr="1">4</operation>
            <operation class="INT_glob_var" type="add">1</operation>
        </loop>
    </procedure>
</app>
```

Slika 3.11: Isječak iz primjera profila aplikacije *AES* - procedura *KeyExpansion*

3.3.3 Algoritam za procjenu trajanja izvođenja

Nakon što su dobiveni profili aplikacije i platforme, zadnji korak je povezivanje oba profila kako bi se izračunala procjena trajanja izvođenja. Algoritam za taj postupak je prikazan u pseudokôdu na slici 3.12.

Za svaku proceduru *procedure* unutar profila aplikacije *applicationProfile* u petlji se prolazi po svim elementima procedure *element*. Za svaki *element* se poziva funkcija *estimate* kojoj se predaje *element* i koja računa procjenu trajanja izvođenja za taj *element*. Nakon toga, za slučaj kada je *element* tipa *operation*, potrebno je broj operacija iste vrste u nizu pohraniti u varijablu *seqstat_len* i ako je $seqstat\_len > 1$, preskočiti *seqstat_len - 1* sljedećih operacija. Razlog tome je što je u funkciji *estimate*, koja je pozvana u prethodnom koraku, već određeno trajanje za taj cijeli niz operacija.

U funkciji *estimate* prvo se provjerava tip *elementa* prema popisu elemenata u tablici 3.8. U slučaju kada je *element* tipa *operation*, najprije se određuje broj operacija iste vrste u nizu i pohranjuje u varijablu *seqstat_len*. Zatim se poziva funkcija *calculate_op_duration* koja izračunava procjenu trajanja izvođenja elementa *operation*. U slučaju kada je *element* tipa *branch*, funkcija *estimate* se poziva za istiniti dio - *truebody* i neistiniti dio - *falsebody t_count* odnosno *f_count* broj puta. Za slučaj kada je *element* tipa *loop*, funkcija *estimate* se poziva rekurzivno za svaki *element* u petlji.

Unutar funkcije *calculate_op_duration* se za operaciju predanu kao argument *element* najprije dohvaćaju informacije iz profila platforme u kojem je definirano trajanje izvođenja svih elementarnih operacija. Na temelju sljedećih atributa *element*a iz profila aplikacije: *class*, *type*, *index_type* i *dim*, pronalazi se čvor u XML strukturi profila platforme koji sadrži podatke za zadanu vrstu operacije. Taj čvor se pohranjuje u varijablu *operation*. Zatim se iz aplikacijskog profila dohvaća vrijednost čvora *element* koja predstavlja broj operacija u nizu unutar izraza i ta vrijednost se pohranjuje u varijablu *seqop_len*. Nadalje slijedi izračun procjene trajanja izvođenja za *element*. Ako je $seqstat\_len > 1$, unutar čvora *operation* se dohvaća čvor-dijete pod nazivom *seqstat* koji ima atribut *nr* najbliži vrijednosti *seqstat_len*. Na primjer, ako je $seqstat\_len = 7$, a profil platforme sadrži unose za duljine nizova 1, 5 i 10, uzet će se vrijednost za duljinu "5". Dohvaćeni čvor-dijete se u pseudokôdu sprema u varijablu *node*. Zatim se provjerava da li *element* ima neki od sljedećih atributa: *mod, add_nr, mul_nr*. Ako ima, traži se čvor *mod*, dijete čvora *node*, čiji atribut *type* ima vrijednost jednaku nazivu atributa *element*-a i dohvaća se njegova vrijednost. Ako nema spomenutih atributa, dohvaća se vrijednost čvora-

Algorithm 1 Procjena trajanja izvođenja

```
for all procedure in applicationProfile do
    for all element in procedure do
        execution_time_est+ = ESTIMATE(element)
        if element is operation then
            seqstat_len = odredi broj operacija u nizu iste vrste kao element
            preskoči sljedećih seqstat_len − 1 operacija
        end if
    end for
end for
function ESTIMATE(element)
    timing_estimate = 0
    if element is operation then
        seqstat_len = odredi broj operacija u nizu iste vrste kao element
        timing_estimate =CALCULATE_OP_DURATION(element, seqstat_len)
    else if element is branch then
        for all element in truebody do
            timing_estimate+ = t_count*ESTIMATE(element)          ▷ truebody count
        end for
        for all element in falsebody do
            timing_estimate+ = f_count*ESTIMATE(element)          ▷ falsebody count
        end for
    else if element is loop then
        for all element in loop do
            timing_estimate+ = ESTIMATE(element)
        end for
        timing_estimate∗ = count                                  ▷ loop count
    end if
    return timing_estimate
end function
function CALCULATE_OP_DURATION(element, seqstat_len)
    operation = dohvati iz profila platforme (element)
    seqop_len = dohvati vrijednost (element)
    if seqstat_len > 1 then
        node= dohvati čvor dijete(operation, "seqstat", seqstat_len)
        if element sadrži (mod || add_nr || mul_nr) then
            duration = skaliraj(dohvati vrijednost(node, "mod", seqstat_len))
        else
            duration = skaliraj(dohvati vrijednost(node, "base", seqstat_len))
        end if
    else if seqop_len > 1 then
        node= dohvati čvor dijete(operation, "seqop", seqop_len)
        duration = skaliraj(dohvati vrijednost(node, "base", seqop_len))
    else
        node= dohvati čvor dijete(operation, "single")
        if element sadrži (mod || add_nr || mul_nr) then
            duration = dohvati vrijednost(node, "mod")
        else
            duration = dohvati vrijednost(node, "base")
        end if
    end if
    return duration
end function
```

Slika 3.12: Pseudokôd algoritma procjene trajanja izvođenja

djeteta *base*. Ta vrijednost se skalira s obzirom na duljinu niza izraza *seqstat_len* i vraća kao povratna vrijednost funkcije *calculate_op_duration*. Ako je *seqstat_len = 1*, ali postoji više operacija u jednom izazu (*seqop_len* > 1), unutar čvora *operation* se dohvaća čvor-dijete pod nazivom *seqop* koji ima atribut *nr* najbliži vrijednosti *seqop_len*. Dohvaćeni čvor-dijete se pohranjuje u varijablu *node*. Zatim se dohvaća se vrijednost čvora-djeteta *base*. Ta vrijednost se skalira s obzirom na duljinu niza izraza *seqop_len* i vraća kao povratna vrijednost. U slučaju kada je *seqstat_len = 1* i *seqop_len = 1*, unutar čvora *operation* se dohvaća čvor-dijete pod nazivom *single*. Dohvaćeni čvor-dijete se pohranjuje u varijablu *node*. Za slučajeve kada postoji više operacija unutar jednog izraza, ne uračunava se učinak atributa modifikatora operacija i indeksa polja kao što je i navedeno u poglavlju 3.3.1. U sljedećem koraku se provjerava da li *element* ima neki od sljedećih atributa: *mod, add_nr, mul_nr*. Ako ima, dohvaća se čvora *mod*, djeteta čvora *node*, čiji atribut *type* ima vrijednost jednaku nazivu atributa *element*-a. U slučaju da *element* nema spomenute atribute, dohvaća se vrijednost čvora-djeteta *base* i ta vrijednost se vraća kao povratna vrijednost funkcije.

3.4 Ispitna okolina i rezultati

U ovom poglavlju prikazana je evaluacija točnosti procjene trajanja izvođenja aplikacije korištenjem metode ELOPS-EM. Odabrane su tri aplikacije i tri konfiguracije platforma iz stvarnog svijeta, s karakteristikama opisanim u nastavku poglavlja. Za svaki ispitni slučaj je najprije napravljena procjena vremena izvođenja, zatim je izmjereno stvarno vrijeme izvođenja te je konačno usporedbom procijenjenog sa izmjerenim vremenom određena pogreška procjene.

3.4.1 Ispitna okolina

Za ciljnu ispitnu platformu odabrana je *Xilinx Zynq ZC706* evaluacijska platforma [8] budući da ona pruža mogućnosti za razvoj i evaluaciju mnoštva različitih dizajna za Zynq®-7000 XC7Z045-2FFG900C AP SoC. ZC706 platforma sadrži mnoge značajke karakteristične za ugradbene računalne sustave, kao npr. DDR3 SODIMM i komponentu memoriju (1GB), četverostruku PCI Express® sabirnicu, Ethernet PHY, općenamjenski I/O i dva UART sučelja. ZC706 platforma je re-konfigurabilna što znači da je moguće sintetizirati različite vrste procesora i memorijskih arhitektura za izvođenje eksperimenata. U ovom slučaju korištene su tri različite konfiguracije procesor-memorija koje su čest izbor za izgradnju brojnih niskoenerget-

skih, termalno ograničenih i cjenovno-osjetljivih uređaja (npr. pametnih telefona, digitalne TV, te potrošačkih i industrijskih aplikacija koje pruža Internet stvari - eng. Intenet of Things (IoT)):

1. *MB1* - MicroBlaze™, 32-bitni procesor temeljen na arhitekturi RISC Harvard [42] sa sljedećom konfiguracijom: cjevovod s 5 stupnjeva, sklopovsko množilo, posmačni registar (engl. *barrel shifter*) i jedinica za operacije s pomičnim zarezom (engl. *floating-point unit - FPU*). Radna frekvencija procesora je 200 MHz. Procesor je direktno povezan s BRAM memorijom na istom FPGA čipu kapaciteta 128KB koja se koristi za pohranu instrukcija i podataka koje koristi procesor.
2. *ARM1* - Jedna jezgra ARM Cortex-A9 procesora koji je također 32-bitni procesor temeljen na arhitekturi RISC Harvard [43]. Radna frekvencija iznosi 667MHz s priručnom memorijom kapaciteta 32 KB na razini L1 i 512 KB na razini L2. Procesor pohranjuje instrukcije i podatke u glavnoj memoriji tipa DDR3 SDRAM radne frekvencije 533 MHz.
3. *ARM2* - Procesor je istih karakteristika kao i u konfiguraciji *ARM1*, ali se instrukcije pohranjuju u DDR3 SDRAM radne frekvencije 533 MHz, a podaci u BRAM memoriji kapaciteta 128KB i radne frekvencije 200 MHz implementiranoj na FPGA čipu koji je dio Zynq SoC-a.

Za ispitne aplikacije odabrani su jedan kriptografski i jedan multimedijski algoritam:

1. algoritam enkripcije *AES* (engl. *The Advanced Encryption Standard*) [11] implementiran u dvije inačice:
   a. *AES_G* - inačica algoritma *AES* u kojoj se podacima koji se dijele između procedura pristupa preko globalnih varijabli,
   b. *AES_P* - inačica algoritma *AES* u kojoj se podaci dijeljeni između procedura šalju kao parametri procedure.
2. algoritam za kompresiju slika *JPEG* - standardna implementacija opisana u [10].

Ove aplikacije su tipični predstavnici skupa aplikacija za koje se najčešće koriste MPSoC sustavi: obrada multimedije, kompresija podataka i enkripcija [44]. U njima su zastupljene pretežno procedure za obradu signala koje podrazumijevaju izvođenje numeričkih operacija nad vektorima i matricama te procedure za pretragu i sortiranje. Zahtijevana količina memorijskog prostora za ove aplikacije je između 50 KB i 250KB što znači da im odgovara veličina priručne memorije tipična za ugradbene procesore te da je očekivano broj pogodaka znatno veći od broja

promašaja.

U kontekstu zastupljenosti elementarnih operacija, ovaj skup aplikacija obuhvaća sve vrste elementarnih operacija sa svih pod-vrstama. U aplikaciji AES_G su prisutne operacije iz sva 4 temeljna skupa elementarnih operacija - točni udjeli su navedeni u tablici 3.9. Operacije s poljima različitih vrsta indeksa su također prisutne: indeks polja je zadan kao jedna varijabla u 25% slučajeva, kao konstantna vrijednost u 9% slučajeva te kao složeni indeks računat na temelju vrijednosti dvije ili više varijabli u oko 15% slučajeva. U aplikaciji AES_P je implementacija algoritma AES modificirana u odnosu na aplikaciju AES_G na način da su sve globalne varijable pretvorene u lokalne varijable glavne (engl. *main*) procedure koje se onda šalju kao parametri procedurama koje se pozivaju iz glavne procedure. Ova modifikacija je napravljena kako bi se u evaluaciju uključile i elementarne operacije s operandima parametrima procedura koje u nisu prisutne u aplikaciji AES_G. Aplikacija JPEG također sadrži operacije iz sva 4 temeljna skupa elementarnih operacija s udjelima navedenim u tablici 3.9. Operandi procedura su lokalne varijable i parametri procedura [e].

Tablica 3.9: Zastupljenost temeljnih skupova elementarnih operacija u ispitnim aplikacijama

	Cjelobrojne operacije	Operacije s pomičnim zarezom	Logičke operacije	Memorijske operacije
AES_G	1%	-	34	65%
AES_P	1%	-	34	65%
JPEG	43%	5%	31%	21%

3.4.2 Ispitni slučajevi

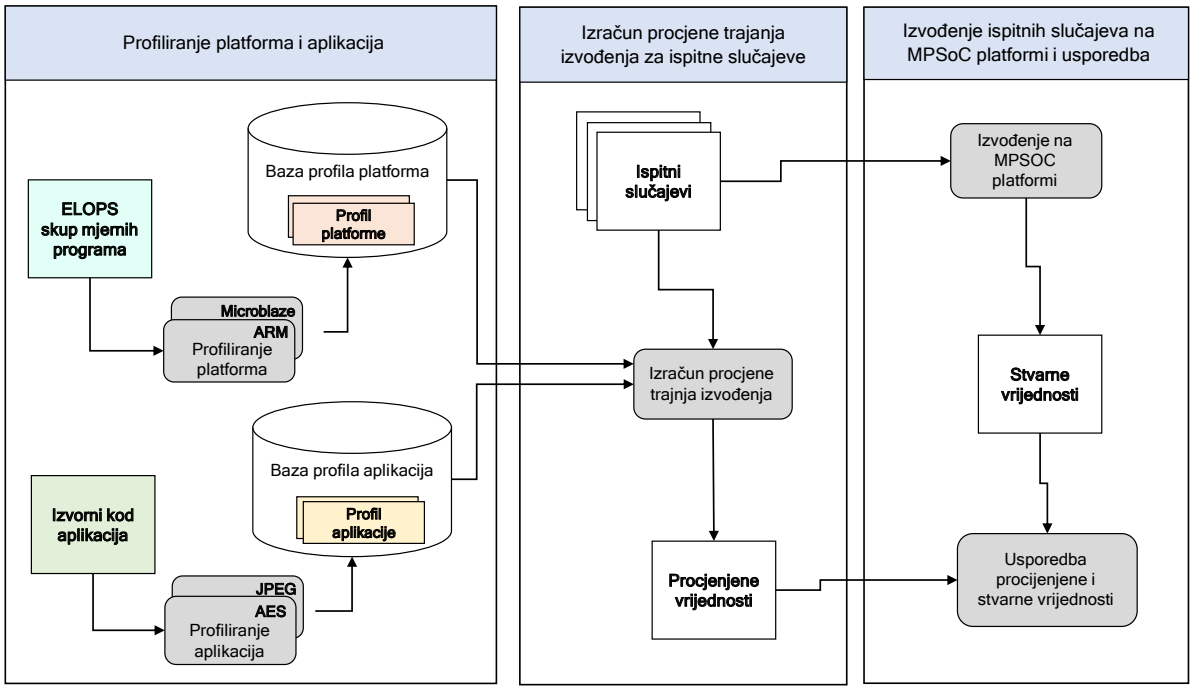
Sve tri aplikacije su ispitane za sve tri konfiguracije platformi i za razine optimizacije O0, O1 i O2. Nadalje, budući da se sve tri aplikacije sastoje od niza procedura koje se pozivaju unutar aplikacije jedan ili više puta, za svaku aplikaciju je ispitana točnost procjene za izvođenje cijele aplikacije te zasebno za svaku pojedinu proceduru od koje se aplikacija sastoji. Tako je za aplikacije AES_G i AES_P uz cjelokupno izvođenje zasebno ispitano i 5 procedura od kojih se sastoje: *KeyExpansion*, *AddRoundKey*, *SubBytes*, *ShiftRows* i *MixColumns*. U slučaju

[e] Izvorni kôd ispitnih aplikacija je dostupan na `https://gitlab.com/Frid/ELOPS`

aplikacije JPEG uz cjelokupno izvođenje aplikacije zasebno je ispitano i 6 procedura od kojih se sastoji: *CreateBlocks*, *Shift*, *DCT*, *ZigZag*, *HuffEncode* i *CreateImage*. To ukupno čini 135 ispitnih slučajeva. Aplikacije AES_G i AES_P su ispitane za ulazne podatke veličine 32 bajta, a aplikacija *JPEG* je ispitana na slici *Lenna* veličine 11KB.

Tijek ispitivanja je prikazan na slici 3.13. Najprije su profilirane sve aplikacije i sve konfiguracije platformi na način kako je opisano u poglavlju 3.3. Zatim je za svaki ispitni slučaj, na temelju profila, izračunata procjena trajanja izvođenja. Nakon toga je svaki ispitni slučaj izveden na platformi kako bi se izmjerila stvarna vremena izvođenja. Mjerenje trajanja izvođenja cijele aplikacije je izvedeno korištenjem sistemskog brojača čija je vrijednost očitana prije i nakon završetka izvođenja cijele aplikacije, a razlika iznosa je uzeta kao stvarno trajanje izvođenja. Mjerenje trajanja izvođenja pojedinih procedura unutar aplikacija je izvedeno tako da je u izvorni kod aplikacije ubačeno očitavanje vrijednosti sistemskog vremenskog brojača prije i nakon poziva procedure, a razlika očitanih iznosa je uzeta kao stvarno trajanje izvođenja te procedure. Pogreška procjene je izračunata prema formuli:

$$Pogreska = \frac{Procjenjeno\_vrijeme - Stvarno\_vrijeme}{Stvarno\_vrijeme} * 100\% \tag{3.2}$$



Slika 3.13: Tijek provedbe ispitivanja

Razina optimizacije O3 nije razmatrana budući da je razina O2 još uvijek preporučena od većine proizvođača ugradbenih sustava jer je sigurna po pitanju potencijalno netočnog redoslijeda izvođenja ako kôd nije napisan točno prema C standardu [6].

Tablica 3.10 sadrži usporedni prikaz procjenjenog i stvarno izmjerenog trajanja izvođenja za aplikaciju *AES_G* za sve tri konfiguracije platformi i razine optimizacije O0 - O2. Prvih pet stupaca u tablici predstavlja svaku od procedura od kojih se AES enkripcija sastoji, a posljednji stupac *Total* sadrži podatke za izvođenje cijele aplikacije nad jednim skupom podataka veličine 32 bajta. Retci tablice predstavljaju konfiguraciju platforme i razinu optimizacije te je u svakom retku navedeno:

1. *Procjena* - procijenjeno vrijeme izvođenja,
2. *Stvarno* - stvarno izmjereno vrijeme izvođenja,
3. *Pogreška* - postotak pogreške procijenjene vrijednosti u odnosu na stvarno izmjerenu.

Ovdje je nužno napomenuti da je trajanje izvođenja jednog poziva procedura *AddRoundKey, SubBytes, ShiftRows* i *MixColumns* mjereno na ponešto drukčiji način nego što je ranije opisano. Budući da je trajanje izvođenja jednog poziva procedura na odabranim konfiguracijama iznimno kratko, reda veličine 10^6 sekundi, trajanje izvođenja je u ovim slučajevima mjereno tako što se procedura uzastopno izvela u petlji 1000 puta, a vrijednost sistemskog vremenskog brojača je očitana prije i nakon završetka cijele petlje. Dobivena vrijednost je zatim podijeljena s 1000 i rezultat je uzet kao stvarno izmjereno vrijeme izvođenja.

Prosječna pogreška je oko 5% s minimumom ispod 1% i maksimumom ispod 16%. Ako se zasebno gleda izvođenje cijele aplikacije, navedeno u stupcu *Total*, prosječna pogreška je nešto manja i iznosi oko 3.5%, minimalna pogreška je 0.2% a maksimalna 7%.

Tablica 3.10: Usporedba procijenjenog i stvarnog vremena izvođenja za *AES_G*

			KeyExpansion	AddRoundKey	SubBytes	ShiftRows	MixColumns	Total
MB-O0	Procjena	[s]	5.81E-05	4.71E-06	5.95E-06	5.20E-07	5.60E-06	4.33E-04
	Stvarno	[s]	5.88E-05	4.98E-06	5.47E-06	5.05E-07	5.57E-06	4.08E-04
	Pogreška	[%]	-1.19	-5.40	8.76	2.97	0.54	6.13
MB-O1	Procjena	[s]	1.77E-05	1.38E-06	1.46E-06	2.80E-07	1.62E-06	1.22E-04
	Stvarno	[s]	1.72E-05	1.35E-06	1.41E-06	2.55E-07	1.48E-06	1.14E-04
	Pogreška	[%]	2.91	2.22	3.55	9.80	9.46	7.02
MB-O2	Procjena	[s]	1.83E-05	1.38E-06	1.38E-06	2.00E-07	1.47E-06	1.17E-04
	Stvarno	[s]	1.76E-05	1.42E-06	1.39E-06	1.90E-07	1.47E-06	1.15E-04
	Pogreška	[%]	3.98	-2.82	-0.72	5.26	0.00	1.74
ARM1-O0	Procjena	[s]	1.42E-05	1.01E-06	1.23E-06	8.28E-07	1.81E-06	1.03E-04
	Stvarno	[s]	1.38E-05	1.11E-06	1.18E-06	8.10E-07	1.65E-06	9.89E-05
	Pogreška	[%]	2.90	-9.01	4.24	2.22	9.70	4.15
ARM1-O1	Procjena	[s]	2.33E-06	1.67E-07	1.73E-07	3.93E-08	2.81E-07	1.68E-05
	Stvarno	[s]	2.29E-06	1.71E-07	1.58E-07	3.75E-08	2.82E-07	1.78E-05
	Pogreška	[%]	1.75	-2.34	9.49	4.80	-0.36	-5.62
ARM1-O2	Procjena	[s]	1.93E-06	1.35E-07	1.73E-07	4.60E-08	2.21E-07	1.46E-05
	Stvarno	[s]	2.02E-06	1.35E-07	1.58E-07	4.95E-08	2.11E-07	1.50E-05
	Pogreška	[%]	-4.46	0.00	9.49	-7.07	4.74	-2.67
ARM2-O0	Procjena	[s]	3.19E-04	2.37E-05	3.01E-05	3.93E-06	3.22E-05	2.28E-03
	Stvarno	[s]	3.17E-04	2.65E-05	2.83E-05	3.79E-06	3.07E-05	2.22E-03
	Pogreška	[%]	0.63	-10.57	6.36	3.69	4.89	2.70
ARM2-O1	Procjena	[s]	9.79E-05	5.18E-06	5.14E-06	2.57E-06	5.66E-06	4.99E-04
	Stvarno	[s]	9.96E-05	5.36E-06	5.32E-06	2.46E-06	5.21E-06	5.00E-04
	Pogreška	[%]	-1.71	-3.36	-3.38	4.47	8.64	-0.20
ARM2-O2	Procjena	[s]	9.79E-05	5.18E-06	5.14E-06	2.57E-06	5.66E-06	4.99E-04
	Stvarno	[s]	9.51E-05	5.24E-06	5.31E-06	3.04E-06	5.28E-06	5.05E-04
	Pogreška	[%]	2.94	-1.15	-3.20	-15.46	7.20	-1.19

Rezultati za drugu implementaciju AES algoritma - *AES_P* navedeni su u tablici 3.11. Korištene su iste tri konfiguracije platformi kao i u prethodnom slučaju. Mjerenje trajanja izvođenja pojedinih procedura je izvedeno na isti način kako je opisano ranije za AES_G. Prosječna pogreška iznosi oko 6% s minimumom ispod 1% i maksimumom ispod 16%. Ako se zasebno

gleda izvođenje cijele aplikacije, prikazano u stupcu *Total*, prosječna pogreška je podjednaka i iznosi oko 6.5%, minimalna pogreška je 0.4% a maksimalna 10.5%.

Tablica 3.11: Usporedba procijenjenog i stvarnog vremena izvođenja za *AES_P*

			KeyExpansion	AddRoundKey	SubBytes	ShiftRows	MixColumns	Total
MB-O0	Procjena	[s]	6.69E-05	5.73E-06	6.51E-06	8.24E-07	6.14E-06	4.74E-04
	Stvarno	[s]	7.25E-05	5.64E-06	5.88E-06	8.80E-07	6.58E-06	4.72E-04
	Pogreška	[%]	-7.72	1.6	10.71	-6.4	-6.69	0.42
MB-O1	Procjena	[s]	1.81E-05	1.38E-06	1.47E-06	2.70E-07	1.48E-06	1.03E-04
	Stvarno	[s]	1.78E-05	1.37E-06	1.44E-06	2.50E-07	1.46E-06	1.15E-04
	Pogreška	[%]	1.68	0.73	2.08	8	1.37	-10.44
MB-O2	Procjena	[s]	1.74E-05	1.10E-06	1.22E-06	1.79E-07	1.29E-06	1.00E-04
	Stvarno	[s]	1.56E-05	1.11E-06	1.08E-06	1.80E-07	1.26E-06	9.23E-05
	Pogreška	[%]	11.54	-0.9	12.96	-0.56	2.38	8.34
ARM1-O0	Procjena	[s]	2.10E-05	1.30E-06	1.46E-06	3.09E-07	2.35E-06	1.34E-04
	Stvarno	[s]	2.23E-05	1.50E-06	1.54E-06	3.02E-07	2.53E-06	1.47E-04
	Pogreška	[%]	-5.83	-13.33	-5.2	2.32	-7.12	-8.84
ARM1-O1	Procjena	[s]	3.80E-06	2.43E-07	2.73E-07	5.89E-08	4.97E-07	2.59E-05
	Stvarno	[s]	3.78E-06	2.46E-07	2.51E-07	6.05E-08	4.56E-07	2.84E-05
	Pogreška	[%]	0.53	-1.22	8.77	-2.65	8.99	-8.8
ARM1-O2	Procjena	[s]	3.11E-06	2.22E-07	2.81E-07	6.81E-08	3.60E-07	2.29E-05
	Stvarno	[s]	3.16E-06	2.26E-07	2.62E-07	7.29E-08	3.48E-07	2.40E-05
	Pogreška	[%]	-1.58	-1.77	7.25	-6.58	3.45	-4.58
ARM2-O0	Procjena	[s]	3.41E-04	2.41E-05	2.16E-05	6.27E-06	3.51E-05	2.21E-03
	Stvarno	[s]	3.56E-04	2.69E-05	2.39E-05	5.41E-06	3.32E-05	2.26E-03
	Pogreška	[%]	-4.21	-10.41	-9.62	15.89	5.72	-2.21
ARM2-O1	Procjena	[s]	9.68E-05	5.12E-06	2.94E-06	2.05E-06	5.37E-06	4.37E-04
	Stvarno	[s]	8.86E-05	4.44E-06	2.96E-06	2.09E-06	4.73E-06	4.01E-04
	Pogreška	[%]	9.26	15.32	-0.68	-1.90	13.50	8.24
ARM2-O2	Procjena	[s]	9.63E-05	5.12E-06	2.94E-06	2.21E-06	4.89E-06	4.28E-04
	Stvarno	[s]	8.76E-05	4.44E-06	2.96E-06	2.59E-06	4.82E-06	3.99E-04
	Pogreška	[%]	9.93	15.32	-0.68	-14.70	1.45	7.27

Tablica 3.12 sadrži rezultate za aplikaciju *JPEG*. Točnost procjene trajanja izvođenja je evaluirana za jedno izvođenje svake pojedine procedure od koje se sastoji JPEG algoritam kom-

presije: *CreateBlocks, Shift, DCT, ZigZag, HuffEncode* i *CreateImage* te za cjelovito izvođenje JPEG algoritma za jedan unos - slika "Lenna" veličine 11K.

Tablica 3.12: Usporedba procijenjenog i stvarnog vremena izvođenja za *JPEG*

			CreateBlocks	Shift	DCT	ZigZag	HuffEncode	CreateImage	Total
MB-O0	Procjena	[s]	1.71E-03	6.96E-04	1.56E-03	7.59E-04	1.00E-02	4.35E-04	3.16E-02
	Stvarno	[s]	1.70E-03	7.17E-04	1.41E-03	7.41E-04	9.37E-03	4.61E-04	2.94E-02
	Pogreška	[%]	0.59	-2.93	10.64	2.43	6.72	-5.64	7.48
MB-O1	Procjena	[s]	2.96E-04	1.39E-04	4.28E-04	1.77E-04	2.75E-03	1.29E-04	1.31E-02
	Stvarno	[s]	2.80E-04	1.42E-04	4.40E-04	1.73E-04	2.83E-03	1.38E-04	1.25E-02
	Pogreška	[%]	5.71	-2.11	-2.73	2.31	-2.83	-6.52	4.8
MB-O2	Procjena	[s]	2.17E-04	1.12E-04	2.69E-04	1.50E-04	2.66E-03	1.13E-04	1.22E-02
	Stvarno	[s]	2.06E-04	1.14E-04	2.88E-04	1.45E-04	2.46E-03	1.22E-04	1.15E-02
	Pogreška	[%]	5.34	-1.75	-6.6	3.45	8.1	-7.38	6.09
ARM1-O0	Procjena	[s]	3.38E-04	1.12E-04	2.77E-04	9.09E-05	2.16E-03	7.68E-05	5.26E-03
	Stvarno	[s]	3.43E-04	1.09E-04	2.66E-04	9.41E-05	2.27E-03	7.13E-05	5.22E-03
	Pogreška	[%]	-1.46	2.75	4.14	-3.4	-4.85	7.71	0.77
ARM1-O1	Procjena	[s]	4.08E-05	1.97E-05	7.95E-05	2.01E-05	6.76E-04	1.75E-05	1.40E-03
	Stvarno	[s]	4.42E-05	2.08E-05	7.94E-05	2.11E-05	7.50E-04	1.67E-05	1.44E-03
	Pogreška	[%]	-7.69	-5.29	0.13	-4.74	-9.87	4.8	-2.78
ARM1-O2	Procjena	[s]	3.94E-05	1.97E-05	7.69E-05	1.61E-05	6.57E-04	1.48E-05	1.33E-03
	Stvarno	[s]	4.45E-05	2.06E-05	7.40E-05	1.75E-05	7.28E-04	1.51E-05	1.39E-03
	Pogreška	[%]	-11.46	-4.34	3.92	-8.0	-9.75	-1.99	-4.32
ARM2-O0	Procjena	[s]	8.97E-03	3.04E-03	1.22E-02	4.22E-03	5.98E-02	2.79E-03	1.30E-01
	Stvarno	[s]	8.36E-03	3.05E-03	1.24E-02	3.62E-03	5.88E-02	2.90E-03	1.27E-01
	Pogreška	[%]	7.3	-0.33	-1.67	16.58	1.7	-3.79	2.36
ARM2-O1	Procjena	[s]	1.34E-03	4.49E-04	2.81E-03	4.67E-04	4.71E-03	6.18E-04	1.78E-02
	Stvarno	[s]	1.42E-03	4.49E-04	2.70E-03	4.77E-04	4.23E-03	6.15E-04	1.71E-02
	Pogreška	[%]	-5.63	0	4.07	-2.1	11.35	0.49	4.09
ARM2-O2	Procjena	[s]	1.34E-03	4.53E-04	2.82E-03	4.70E-04	4.64E-03	6.18E-04	1.78E-02
	Stvarno	[s]	1.42E-03	4.56E-04	3.00E-03	4.97E-04	4.17E-03	6.13E-04	1.84E-02
	Pogreška	[%]	-5.63	-0.66	-6.0	-5.43	10.13	0.82	-3.26

Potrebno je napomenuti da je procjena trajanja izvođenja za proceduru *HuffEncode*, koja ostvaruje huffmanovo kodiranje, napravljena koristeći instrumentaciju izvornog kôda i simulaciju izvođenja za dani ulaz. To je učinjeno zato što veličina rezultata kodiranja značajno varira ovisno o ulaznim podacima, što znači da se ulazne slike iste veličine ali različitog sadržaja mogu

znatno razlikovati u konačnoj veličini. Ovo je primjer slučaja kada su mogući različiti putevi unutar jedne procedure i svaki put se može ponoviti različit broj puta ovisno o ulazu. Stoga je bilo nužno dodatno instrumentirati izvorni kôd kako bi se doznalo koji se putevi u proceduri izvode koliko puta. Izvorni kôd je instrumentiran ručno i izveden jednom na PC računalu za iste ulazne podatke - sliku Lenna. Rezultati instrumentacije su korišteni za procjenu trajanja izvođenja za sve konfiguracije. Ako se uzmu u obzir svi ispitni slučajevi za aplikaciju JPEG, prosječna pogreška je oko 5% s minimumom ispod 1% i maksimumom ispod 17%. Ukoliko se promatra samo izvođenje cijele aplikacije odjednom, navedeno u stupcu *Total*, prosječna pogreška iznosi oko 4%, najmanja pogreška je 0.8%, a najveća pogreška iznosi 7.5%.

U tablici 3.13 je na primjeru algoritma JPEG prikazana usporedba pogreške procjene metode ELOPS-EM i ostalih metoda iz poglavlja 3.1. koje su koristile algoritam JPEG za evaluaciju. Metoda ELOPS-EM postiže bolje rezultate u odnosu na sve metode osim one koju su predložili Lin i ostali u [4]. Ta metoda se temelji na simulaciji na razini osnovnih blokova koja, kako je ranije istaknuto, jamči visoku točnost rezultata.

Tablica 3.13: Usporedba pogreške procjene metode ELOPS-EM i ostalih metoda na primjeru algoritma *JPEG*

	Prosječna pogreška	Maksimalna pogreška
ELOPS-EM	4%	7.5%
Nikolov i ostali (2008.) [29]	12%	19%
Oyamada i ostali (2007.) [25]	9%	36%
Cilardo i ostali (2013.) [27]	-	20%
Lin i ostali (2010.) [4]	2%	-

Zaključno, za sve ispitne slučajeve točnost procjene je približno na istoj razini, prosječna pogreška iznosi oko 5%, dok maksimalna pogreška ne prelazi 17%. Gledano samo za izvođenja cijelih aplikacija odjednom, navedenih u stupcu *Total*, prosječna pogreška je podjednaka ukupnoj prosječnoj pogrešci, dok je maksimalna pogreška niža i ne prelazi 10%. Gotovo podjednako su zastupljene procjene u kojima je podcijenjena vrijednost trajanja izvođenja i procjene u kojima je ona precijenjena. U budućem istraživanju bi trebalo pokušati odrediti u kojim točno slučajevima metoda ELOPS-EM daje premalu, a u kojima preveliku vrijednost procjene. Gledano s aspekta razine optimizacije kôda, također nije uočeno odstupanje u točnosti procjene za

različite razine optimizacije.

Nadalje, promatranjem rezultata za konfiguraciju *ARM2*, koja je osjetljivija na utjecaj priručne memorije budući da procesor komunicira sa znatno sporijom memorijom, nije primijećeno odstupanje u razini pogreške u usporedbi s ostale dvije konfiguracije. U slučaju kada predložena metoda ne bi mogla dobro aproksimirati učinke priručne memorije, za ispitne slučajeve s konfiguracijom *ARM2* vrijednost procjene bila bi znatno precijenjena kod pogodaka, a znatno podcijenjena kod promašaja priručne memorije, što ovdje nije primijećeno. No, nužno je ponovno napomenuti da odabrane ispitne aplikacije dobro predstavljaju tipične aplikacije za koje se najčešće koriste MPSoC sustavi. Samim time zahtijevana količina memorijskog prostora dobro odgovara veličini priručne memorije tipične za ugradbene procesore te je očekivano broj pogodaka znatno veći od broja promašaja. To može značiti da je tijekom evaluacije bila mnogo više situacija u kojima su se događali pogodci nego promašaji priručne memorije.

Postignuta razina točnosti je bolja u usporedbi s analitičkim metodama u kojima je prosječna pogreška 17% [25] do 20% [26, 27]. Nešto je lošija u usporedbi s točnosti simulacijskih metoda koje imaju razinu maksimalnu pogreške ispod 10% [4, 5, 6, 7, 24]. No, u odnosu na simulacijske metode, metodu ELOPS-EM karakterizira visoka razina ponovne iskoristivosti profila aplikacije i platforme. Svaka se aplikacija treba profilirati samo jednom i dobiveni se profil može koristiti bez ikakvih izmjena za bilo koju konfiguraciju platforme. Na isti način, svaka vrsta elementa platforme se treba profilirati samo jednom i dobiveni se podaci mogu koristiti za procjenu trajanja izvođenja bilo koje aplikacije. Tako npr. u slučaju izmjena dijelova kôda aplikacije, zbog paralelizacije ili optimizacije algoritma, nije nužno ponovno raditi vremenski zahtjevna mjerenja na ISS-u ili fizičkoj platformi kao što je slučaj kod metoda koje temelje procjenu na osnovnim blokovima. Mogućnost ponovnog korištenja ranije dobivenih profila pridonosi smanjenju napora i vremena potrebnog za dobivanje rane procjene trajanja izvođenja. Metoda ELOPS-EM i postignuti rezultati su objavljeni 2019. godine [45].

Poglavlje 4

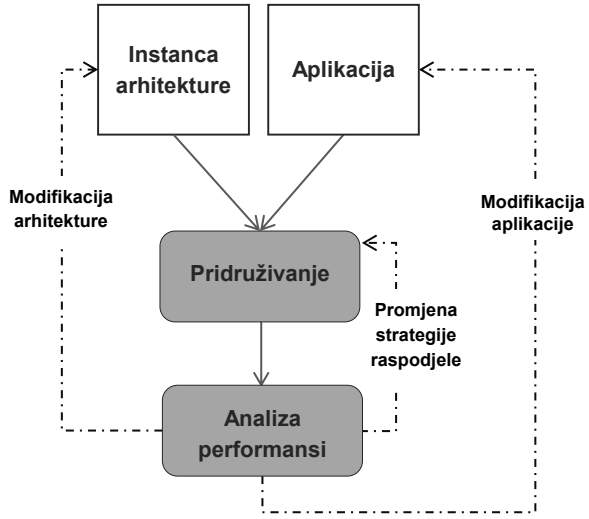
Pretraga prostora oblikovanja

Pretraga prostora oblikovanja - DSE (engl. *design space exploration*) je proces pronalaženja optimalne ili skoro optimalne arhitekture za određenu aplikaciju unutar zadanih ograničenja (engl. *constraints*) kao npr.: ukupno trajanje izvođenja, broj elemenata na platformi, potrošnja energije, cijena itd. To podrazumijeva definiranje točne vrste i broja procesnih, memorijskih i komunikacijskih elementa sklopovske platforme, razdvajanje (engl. *partitioning*) dijelova aplikacije koji će biti implementirani programski od onih koji će biti implementirani sklopovski i pridruživanje (engl. *mapping*) programski implementiranih dijelova aplikacije na elemente platforme [14]. Pod pojmom *dijelovi aplikacije* se najčešće podrazumijevaju procedure koje je pozivaju unutar aplikacije, no to mogu biti i manji dijelovi kôda unutar procedura koji se mogu izvoditi u paraleli i sl.

Pridruživanje se odvija ručno ili automatski te je popraćeno analizom performansi kako bi se utvrdilo udovoljava li sustav svim zahtjevima i poštuje li zadana ograničenja. Sve dok se ne postigne zadovoljavajući rezultat moguće je izvođenje modifikacija na aplikaciji, arhitekturi ili listi pridruživanja dijelova aplikacije po elementima platforme. Jednom kad je postignut zadovoljavajući rezultat prelazi se u fazu implementacije fizičke platforme. Tijek procesa pretrage prostora oblikovanja je ilustriran na slici 4.1.

Primarni naglasak kod DSE-a je na brzom i efikasnom izdvajanju malog skupa najboljih rješenja iz velikog skupa potencijalnih rješenja. Pri tome procjena performansi ne mora biti vrlo precizna budući da se nakon faze DSE-a skup izdvojenih rješenja može realizirati do razine prototipa i napraviti detaljna verifikacija.

Važnost DSE-a posebno je istaknuta kod heterogenih višeprocesorskih sustava budući da oni sadrže različite vrste procesora, memorija i komunikacijskih elemenata koji nisu podjed-



Slika 4.1: Shema procesa pretrage prostora oblikovanja

nako učinkoviti za izvođenje svih tipova aplikacija. No, uz odabir dobre konfiguracije, koja podrazumijeva točno određenu kombinaciju elemenata i pridruživanje aplikacije na te elemente, moguće je postići performanse (npr. brzina izvođenja, potrošnja energije i sl.) superiornije onima koje bi se postigle na homogenoj platformi s istim brojem elemenata. Najveća prepreka u pronalaženju optimalne konfiguracije jest činjenica da zbog prevelikog broja kombinacija različitih elemenata nije moguće izgraditi sve prototipove i na njima mjeriti performanse. Također, alati za razvoj ovakvih sustava su vrlo heterogeni, slabo-povezani i zahtijevaju puno vremena za učenje. Stoga je nužno znatno suziti izbor kandidatskih konfiguracija prije izgradnje prototipa. Kako bi se to postiglo, za procjenu performansi uvode se modeli aplikacije i platforme visoke razine apstrakcije koji zahtijevaju znatno manje vremena i napora za izradu te omogućuju dobivanje rane procjene performansi što značajno ubrzava proces pretrage [23].

No čak ni uz korištenje modela visoke razine apstrakcije, pronalaženje optimalne konfiguracije nije jednostavan zadatak te najčešće nije moguće pronaći rješenje iscrpnom pretragom svih mogućih konfiguracija, nego je nužno koristiti različite heurističke metode. Evolucijski algoritmi su u posljednje vrijeme vrlo široko prihvaćeni kao metoda pretrage prostora oblikovanja. Njihova prednost je što se mogu vrlo lako prilagoditi bilo kojem problemu te omogućuju istovremenu optimizaciju više kriterija. Također, iako ne osiguravaju pronalaženje optimalnog rješenja, u vrlo kratkom vremenskom roku mogu doći do rješenja koja zadovoljavaju zadana

ograničenja.

U ovom poglavlju predlaže se metoda za pretragu prostora oblikovanja sustava modeliranog na visokoj razini prema načelima SLD-a, što podrazumijeva razdvojene modele aplikacije i platforme te odvojen izračun od komunikacije. Model sustava obuhvaća modele aplikacije i platforme koji se grade na osnovu koncepta elementarnih operacija te profila aplikacija i platforme opisanih u prethodnom poglavlju. Ti modeli se koriste u heurističkoj metodi pretrage prostora oblikovanja koja ima za cilj pronalaženje optimalne sheme pridruživanja (engl. *mapping scheme*) dijelova aplikacije na elemente platforme te određivanje redoslijeda izvođenja. Pri tome se procedure pridružuje procesorima, a komunikacijske kanale memorijama.

Metoda se temelji na evolucijskom algoritmu za višekriterijsku optimizaciju NSGA-II [12], a optimiraju se dva kriterija: vrijeme izvođenja i broj elemenata platforme. S obzirom na veliki utjecaj memorijske konfiguracije na performanse cijelog sustava, pretraga prostora oblikovanja obuhvaća i procesore i memorije. Osmišljene su dvije inačice pretrage prostora oblikovanja: istovremena pretraga i dvostupanjska pretraga. U istovremenoj pretrazi, istovremeno se procedure pridružuju procesorima i komunikacijski kanali memorijama. Dvostupanjska pretraga se odvija u dvije iteracije pa se prvo pridružuje procedure procesorima, a zatim komunikacijske kanale memorijama.

Evaluacija je napravljena na umjetno stvorenim modelima i modelima stvarnih aplikacija i platforma. Odabrane stvarne aplikacije su JPEG kompresija i praćenje zrake (engl. *ray tracing*) te platforme Xilinx ZC706 i Adapteva Parallella.

Posebno je razmotren slučaj kada svi procesori nemaju pristup svim memorijama, tj. kada su elementi na platformi rijetko povezani. U takvom slučaju broj nemogućih može znatno premašiti broj mogućih rješenja što problem pretrage prostora oblikovanja čini složenijim. Algoritam istovremene pretrage je variran u dodatne četiri inačice posebno prilagođene takvim platformama, a učinkovitost svake inačice je provjerena na pet posebno stvorenih umjetnih modela.

U nastavku poglavlja najprije je dan pregled najvažnijih radova koji se bave problematikom pretrage prostora oblikovanja heterogenih sustava. Zatim slijedi opis modela sustava i metode pretrage prostora oblikovanja te evaluacija rezultata na umjetnim i stvarnim ispitnim slučajevima. Na kraju poglavlja je razmotren slučaj rijetko povezane platforme.

4.1 Pregled radova

Postojeće metode za pretragu prostora oblikovanja se suštinski mogu podijeliti na dva pristupa. Prvi pristup se fokusira na odabir elemenata od koji će biti izgrađena platforma tako da budu najpogodniji za izvođenje pojedinih dijelova aplikacije. Kôd aplikacije je fiksan, a platforma labava: zadana je vrsta elemenata koji se mogu koristiti, ali ne i njihov broj i međusobna povezanost. Rezultat pretrage prostora oblikovanja je konačna fiksirana definicija platforme i shema pridruživanja (engl. *mapping scheme*) dijelova aplikacije na elemente platforme. Drugi pristup je upravo suprotan: programski kôd aplikacije se prilagođava fiksno zadanoj platformi kroz optimiranje i paralelizaciju dijelova kôda.

Među alatima razvijenim u akademskoj zajednici, dva alata se posebno ističu: *Daedalus* [28] i *MAPS* [30]. *Daedalus* slijedi prvi pristup: dijelovi aplikacije se pridružuju na procesne elemente između kojih postoji samo jedan memorijski element. Postoji i mogućnost naknadne ručne intervencije i odabir alternativnih memorijskih elemenata putem kojih će se izvoditi komunikacija ukoliko je potrebno, tj. ukoliko najbolja ponuđena konfiguracija sustava ne udovoljava zahtjevima. Budući da se radi o NP-teškom problemu, za pretragu prostora rješenja se koristi heuristički evolucijski algoritam SPEA-2 [46]. Alat *MAPS* slijedi drugi pristup pa se najprije aplikacija prilagođava zadanoj platformi koristeći napredne opcije MAPS prevoditelja, a zatim se radi pridruživanje i raspoređivanje koristeći standardne postupke kao što su: prioritetno raspoređivanje s prekidima ili bez prekida, Round-Robin i sl [47]. Oba alata temelje se na načelima SLD-a te koriste zasebne modele aplikacije i platforme na visokoj razini apstrakcije.

Većina radova iz područja se uglavnom fokusira na odabir elemenata od koji će biti izgrađena platforma te pronalazak optimalne sheme pridruživanja. Rani radovi su razvijali vlastite heuristike kako bi dobili shemu pridruživanja što bližu optimalnoj u prihvatljivom vremenskom periodu [13, 20]. U tim heuristikama isključivo se radi pridruživanje aplikacije na pojedine procesore, dok se uopće ne razmatra pridruživanje komunikacijskih kanala na memorije. U novijim radovima su zastupljenije metode raspoređivanja zasnovane na unaprijeđenim metodama cjelobrojnog programiranja - ILP (engl. *integer linear programming*) [48] ili evolucijskim algoritmima kao što su SPEA-2 i NSGA-II [49] koji omogućuju višekriterijsku optimizaciju.

Budući da je već i samo pridruživanje aplikacije na procesore NP-težak problem, većina radova se primarno bavi rješavanjem tog problema, a pridruživanje komunikacijskih kanala je sekundarno. Alati *Daedalus* i *MAPS*, također slijede ovaj pristup. U oba slučaja, proces generiranja sheme pridruživanja ne uzima u obzir moguće postojanje više komunikacijskih puteva

između procesora (više različitih memorija na koje procesori mogu biti spojeni, tj. preko kojih mogu razmjenjivati podatke). Memorijska konfiguracija i shema alokacije podataka se radi ručno u kasnijoj fazi oblikovanja i implementacije.

S druge strane, neki autori se fokusiraju isključivo na alokaciju memorijskih resursa tj. raspodjelu (engl. *partitioning*) programskog kôda i podataka u memoriji bez integracije u proces pretrage prostora oblikovanja. Verma i ostali [50] bave se ugradbenim sustavima koji imaju priručnu memoriju i SPM (engl. *scratchpad memory*). Oni predlažu metodu temeljenu na cjelobrojnom programiranju u kojoj se koristi statička analiza toka podataka za generiranje modela pogodaka priručne memorije. Time se dobivaju rješenja za alokaciju SPM-a koja uzimaju u obzir i model priručne memorije. Za predstavljenu pohlepnu strategiju pod nazivom *Cache Aware Scratchpad Allocation (CASA)* tvrde da postižu prosječno smanjenje potrošnje energije od 8-29% u usporedbi s ranije objavljenim tehnikama za *Mediabench* ispitni skup [51]. Ovaj pristup ne uzima u obzir topologiju arhitekture i propusnost (engl. *bandwith*). Falk and Kleinsorge [52] koriste model temeljen na ILP-u kako bi minimizirali vrijeme izvođenja u najgorem slučaju - WCET (engl. *worst-case execution time*) za memorijske sustave sa SPM-om. Oni uzimaju u obzir postojanje samo jedne scratchpad memorije. Na sličan način Suhendra i ostali [53] prezentiraju pristup koristeći statičku analizu toka podataka koja generira i rješava ILP model za minimiziranje WCET-a pri alokaciji SPM-a te heuristiku za pronalaženje dobrih rješenja. Svi ovi modeli ne uzimaju u obzir složenije topologije s više od jedne memorije i procesora te različitom propusnosti na sabirnicama. Model alokacije je vrlo jednostavan: razmatra se jedan po jedan osnovni block (engl. *basic block*) izvornog kôda koji se ili stavlja u SPM ili u glavnu radnu memoriju. Ozturk i ostali [54] predstavljaju nešto cjelovitiji pristup koji uz generiranje memorijske raspodjele analizira i izvorni kôd radi procjene trajanja izvođenja i identifikacije ostalih karakteristika. Korišteni memorijski model je samo djelomično svjestan topologije arhitekture.

Cjelovit pristup u kojem je fokus na integraciji alokacije memorijskih resursa u DSE može se pronaći u sljedećim radovima. Salamy i Ramanujam [55] prezentiraju heuristiku koja integrira raspoređivanje zadataka i particioniranje memorije za aplikacije na heterogenim MPSoC platformama sa scratchpad memorijom. U usporedbi s odvojenim pristupom, integrirani pristup skraćuje vrijeme trajanja izvođenja za oko 12% u prosjeku (za neke aplikacije i do 47%). No, autori razmatraju vrlo jednostavnu memorijsku arhitekturu s jednom SPM podijeljenom između dva procesora te jednom glavnom radnom memorijom. Jovanovic i ostali [56] razmatraju

stvarne heterogene MPSoC sustave s različitim vrstama procesora i memorija, ali memorijska alokacija se odvija nakon dodjele zadataka na procesore te ne uzima u obzir mogućnost istovremene evaluacije para procesor-memorija. Iako autori spominju poboljšanje od oko 18% za standardne ispitne slučajeve (FIR, IIR, MPEG4, kompresija i sl.), metoda optimizacije se temelji na ILP-u što znači da je za složenije probleme vrijeme dolaska do rješenja vrlo dugo, a moguće i beskonačno. Goens i ostali [49] koriste mješovito cjelobrojno programiranje - MILP (engl. *mixed integer linear programming*) za optimizaciju alokacije podataka u memorije na heterogenoj platformi. Ponovno, shema alokacije podataka u memorije se generira nakon alokacije zadataka na procesor. Metoda je fokusirana prvenstveno na aplikacije sa strujanjem podataka (engl. *streaming applications*) pa je stoga cilj optimizacije nije pronalaženje najboljeg, već onog rješenja koje će zadovoljiti zadana ograničenja. Ovakva relaksacija problema skraćuje vrijeme izvođenja MILP-a, ali, prema navodima autora, za veći memorijski prostor predložena metoda nije pronašla rješenja ni nakon 4 dana izvođenja na super-računalu.

Unatoč brojnim istraživanjima, još uvijek ne postoji jedinstvena metoda prikladna za pretragu prostora oblikovanja MPSoC sustava. Zbog velike raznolikosti tih sustava, prvenstveno u vidu arhitekture procesora i memorija, problem pronalaženja optimalne sheme pridruživanja i rasporeda postaje NP-težak. Problem dodatno usložnjava i činjenica da u se u MPSoC sustavima osim općenamjenskim procesora, mogu naći i razni sklopovski akceleratori na kojima se mogu izvesti samo neki dijelovi aplikacije. Nadalje, komunikacija između procesora ne ide direktno nego preko memorija s ograničenim kapacitetom i brojem priključaka, a moguć je i slučaj da procesori međusobno nisu potpuno povezani. Time se stvara veliki broj kombinacija od kojih samo manji dio čine izvediva rješenja. Većina autora je stoga ograničila i olakšala problem svodeći ga na sustav u kojem se sve zadaće mogu izvesti na svim procesorima, procesori uglavnom komuniciraju putem jedne dijeljene memorijom ili više memorija iste vrste. Konačno, u literaturi vezanoj uz MPSoC sustave nije pronađeno razmatranje problema rijetko povezanog sustava.

4.2 Model sustava

U srži svake metodologije nalaze se modeli u različitim koracima razvojnog procesa. Općenita definicija modela prema [57] glasi: *"Model je pojednostavljenje nekog entiteta, koji može biti fizička stvar ili drugi model. Model sadrži točno one značajke i svojstva modeliranog entiteta*

koje su važne za određenu zadaću. Model je minimalan u odnosu na neku zadaću ako ne posjeduje nijedne druge značajke osim onih koje su važne za tu zadaću."

Modeli na različitim razinama apstrakcije pružaju podlogu za analizu, sintezu i verifikaciju. Koncepti i tehnike koji se primjenjuju kod modeliranja imaju veliki utjecaj na kvalitetu, točnost i brzinu dobivanja rezultata. Iz tog razloga je nužno pri izradi modela dobro odrediti razinu i organizaciju detalja koji će biti prikazani te pravila organizacije i transformacije kako bi se moglo adekvatno specificirati zahtjeve i ograničenja, dobiti valjana i relevantna opažanja te primijeniti alati za automatizaciju procesa [13].

U oblikovanju ugradbenih sustava koriste se tri vrste modela: specifikacijski, transakcijski - TLM (engl. *transaction-level model*) i na razini ciklusa - CAM (engl. *cycle-accurate model*). Specifikacijski modeli se zadaju na najvišoj razini apstrakcije, a za opis se koriste računski modeli - MoC (engl. *model of computation*) i ograničenja na performanse arhitekture [58]. Model TLM sadrži više detalja o samoj strukturi aplikacije i arhitekture, a najvažnija značajka je razdvajanje komunikacije (engl. *communication*), tj. razmjene podataka između dijelova aplikacije od izračuna (engl. *computation*). Izračun je predstavljen procesima, a komunikacija kanalima koji međusobno povezuju procese [32]. Model CAM sadrži detalje potrebne za implementaciju sustava. Arhitektura je opisana na razini fizičkog sloja (komponente, priključnice, sabirnice itd.), detaljno su definirani komunikacijski protokoli, a aplikacija je zadana na razini asemblerskog kôda.

Metodologije koje se zasnivaju na SLD-u tipično započinju s modeliranjem i simulacijom mogućih komponenti sustava i njihovih interakcija u vrlo ranoj fazi oblikovanja [22]. Ključni koncept je specifikacijski model sustava koja uključuje zasebne modele arhitekture i aplikacije (osnovna ponašajna specifikacija). Time se definira ukupan prostor oblikovanja i mogućih konačnih rješenja koja se pretražuje kroz iterativan proces implementacije, verifikacije i redefiniranja specifikacije [23]. Aplikacija se zadaje računskim modelom koji opisuje ponašanje sustava pomoću procesa i stanja. Sklopovska arhitektura se opisuje pomoću komponenti od kojih se sastoji i za koje su zadani skup parametara i način povezivanja. Konačno je zadan i skup ograničenja u vidu cijene, potrošnje energije, zauzeća prostora i sl. koje implementirani sustav mora zadovoljiti.

Modeli na razini sustava definirani su na visokoj razini apstrakcije te predstavljaju ponašanje aplikacije, značajke arhitekture i odnos (pridruživanje, particioniranje i sl.) između aplikacije i arhitekture. Odabir visoke razine apstrakcije pri modeliranju prvenstveno minimizira napor i

vrijeme koje je potrebno uložiti u izradu modela. Budući da sadrže vrlo malo detalja, takve je modele moguće vrlo brzo simulirati što omogućava brzu verifikaciju dizajna i procjenu performansi, potrošnje energije i cijene pri pretrazi prostora rješenja u ranim fazama oblikovanja kada je prostor potencijalnih oblikovnih rješenja jako velik. Preciznost performansi procijenjenih na temelju ovakvih modela nije visoka - pogreška je veća od 10%, no u ranim fazama oblikovanja je znatno važnije brzo suziti prostor mogućih rješenja, a onda je kasnije moguće precizno evaluirati mali skup odabranih rješenja.

Metoda za pretragu prostora oblikovanja koja je predložena u ovom radu također se oslanja na model sustava karakterističan za SLD pristup u kojem su razdvojeni modeli aplikacije i platforme te izračun od komunikacije. Model sustava gradi se na osnovu koncepta elementarnih operacija te profila aplikacije i platforme opisanih u prethodnom poglavlju. U nastavku poglavlja je detaljnije opisano kako su definirani modeli aplikacije i platforme visoke razine apstrakcije.

4.2.1 Model aplikacije

SLD model aplikacije načelno podrazumijeva skup istovremenih procesa organiziranih u hijerarhiju koji rade nad skupom podataka te međusobno razmjenjuju te podatke. Na apstraktnoj razini to se prikazuje pomoću MoC-a [14] što omogućuje lakše razumijevanje ponašanja složenih sustava. Model pojednostavnjuje ponašanje aplikacije otkrivajući samo vršni pogled (krupna granulacija), a svi detalji niže razine su apstrahirani.

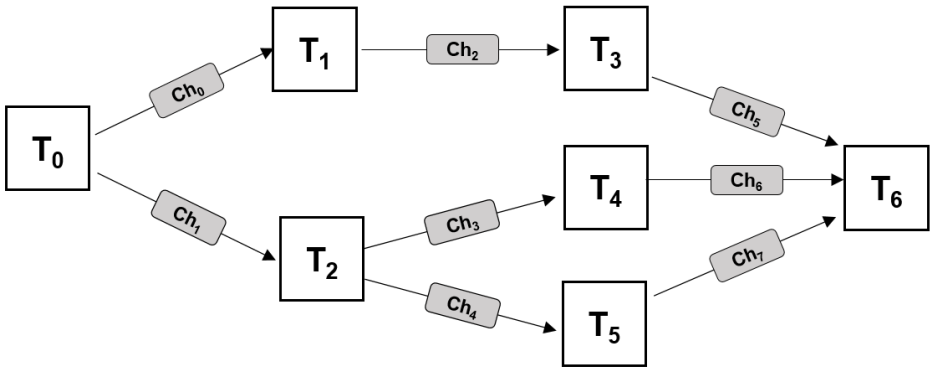
Budući da je većina aplikacija za ugradbene sustave još uvijek dominantno pisana u jezicima C ili C++, koji su imperativni, ne postoji jednoznačan način izrade modela visoke razine apstrakcije za takve aplikacije koji je ujedno i prilagođen za MPSoC platforme gdje je iznimno važno prikazati paralelizam. Zato postoji više različitih kategorija MoC-ova koje je moguće koristiti za prikaz ponašanja aplikacije: modeli stanja (engl. *state-based models*), procesni modeli (engl. *process-based models*) i paralelni modeli (engl. *concurrent models*). U praksi su najčešće zastupljeni modeli koji pripadaju u više od jedne kategorije: većina radova oslanja se na neki od modela koji omogućavaju izražavanje paralelizma na razini zadaća (dijelova aplikacije, najčešće procedure) i eksplicitno prikazuju komunikaciju između zadaća što omogućava lakše pridruživanje aplikacije na platformu. To su najčešće modeli koji omogućavaju prikaz paralelizma tako što je aplikacija specificirana kao skup procesa koji su međusobno povezani kanalima koji predstavljaju tok podataka između procesa. Na taj način vidljiva je međuovisnost

između procesa te mogućnost paralelnog izvođenja dijelova aplikacije, tj. pridruživanja procesa na procesne elemente.

Svi procesni modeli se temelje usmjerenom acikličkom grafu - DAG (engl. *directed acyclic graph*) [59]. DAG je generički model paralelnog programa koji se sastoji od skupa čvorova koji predstavljaju procese, a način na koji su međusobno povezani prikazuje podatkovnu međuovisnost procesa. Definicija DAG-a kao paralelnog modela izračuna glasi:

$$AG = (T, Ch). \tag{4.1}$$

Svaki čvor $T_i \in T$ predstavlja jedan proces (zadaću, engl. *task*), tj. skup instrukcija koje se izvode slijedno bez prekida na istom procesoru. Za svaki čvor T_i definira se skup usmjerenih bridova $Ch_j \in Ch$ koji povezuju čvor T_i s jednim ili više čvorova u grafu i predstavljaju komunikaciju između tih procesa. Neki čvor može imati više ulaza, a tek kad su svi ulazi dostupni, čvor (tj. proces koji on simbolizira) se može izvesti. Nakon izvođenja procesa spremni su izlazi koji se šalju dalje prema čvorovima s kojima je taj čvor povezan. Čvor koji nema prethodnika je *početni čvor*, a čvor koji nema sljedbenika je *završni čvor*. Čvorovi koji nisu međusobno podatkovno međuovisni mogu se izvoditi paralelno. Čvor može imati težinu koja predstavlja trošak izvođenja (broj instrukcija) i označava se kao $w(T_i)$. Brid također može imati težinu koja predstavlja količinu podataka koji se prenose između dva čvora - $w(Ch_j)$. DAG se još ponekad naziva i graf zadataka (engl. *task graph*). Na slici 4.2 je ilustriran primjer DAG-a kojim je modelirana generička aplikacije koja se sastoji od nekoliko procesa međusobno povezanih komunikacijskim kanalima.



Slika 4.2: Primjer DAG-a za generičku aplikaciju koja se sastoji od nekoliko procesa međusobno povezanih komunikacijskim kanalima

Posebne vrste DAG-a koje se osobito koriste u aplikacijama za obradu signala su graf toka podataka - DFG (engl. *Data Flow Graph*) i Kahnove procesne mreže - KPN (engl. *Kahn process networks*). DFG se sastoji od čvorova povezanim usmjerenim bridovima. Čvorovi predstavljaju operacije nad podacima, a bridovi podatkovnu međuovisnost čvorova. Neki bridovi nemaju izvorišni čvor već samo odredišni - oni predstavljaju ulazni tok podataka. Bridovi koji imaju samo izvorišni, ali ne i odredišni čvor, predstavljaju izlazni tok podataka. U DFG-u također nema ciklusa, prikazuje se paralelizam između čvorova, a redoslijed izvođenja se određuje na temelju podatkovnih međuovisnosti. KPN je sličan koncept DFG-u s tom razlikom što su komunikacijski kanali predstavljeni kao neograničeni red po redoslijedu dospijeća - FIFO (engl. *first-in first-out*). Svaki red FIFO može imati samo jedan ulaz i samo jedan izlaz, a proces čitanja iz reda je blokirajući te je potrebno uvesti pravila sinkronizacije.

Radni okviri Daedalus i MAPS koriste KPN-ove za specifikaciju aplikacije. Model aplikacije temeljen na KPN-u omogućava jednostavno izražavanje paralelizma unutar aplikacije, no potrebne su dodatne modifikacije kako bi se jasno i točno iskazala podatkovna međuovisnost i struktura komunikacije (npr. ograničavanje neograničenog FIFO spremnika), što svaki alat rješava na svoj način u kasnijim fazama pretrage prostora oblikovanja. Arhitektura platforme opisuje se koristeći XML (ili podinačicu YML – Daedalus) zbog svoje fleksibilnosti koja omogućava relativno jednostavnu parametrizaciju platforme.

Metoda za pretragu prostora oblikovanja opisana u ovom radu koristi DAG kao podlogu za izradu modela aplikacije zbog svoje fleksibilnosti te jednostavne prilagodbe algoritmu optimizacije koji će biti opisan kasnije. Cijela aplikacija je prikazana jednim grafom:

$$APP = (T, Ch) \tag{4.2}$$

Čvorovi grafa, $T_i \in T$, predstavljaju procedure unutar aplikacije i međusobno su povezani usmjerenim bridovima, $Ch_j \in Ch$, prema načinu na koji su procedure međusobno podatkovno ovisne. Struktura grafa se iščitava iz aplikacijskog profila temeljenog na elementarnim operacijama opisanog u poglavlju 3.3.2. - čvorovi u grafu odgovaraju elementima pod nazivom *procedure*. Čvorovi T_i nemaju težinu, a trajanje izvođenja procedura na platformi se zadaje u modelu platforme koji će biti opisan u idućem potpoglavlju. Bridovi grafa Ch_j predstavljaju elementarnu operaciju MEM_BLOCK koja se poziva kako bi se prenijeli podaci između dvije procedure i to samo ako se procedure nalaze na različitim procesorima. Težina bridova $w(Ch_j)$ je količina podataka u KB koju je potrebno prenijeti izvođenjem MEM_BLOCK operacije. Ne-

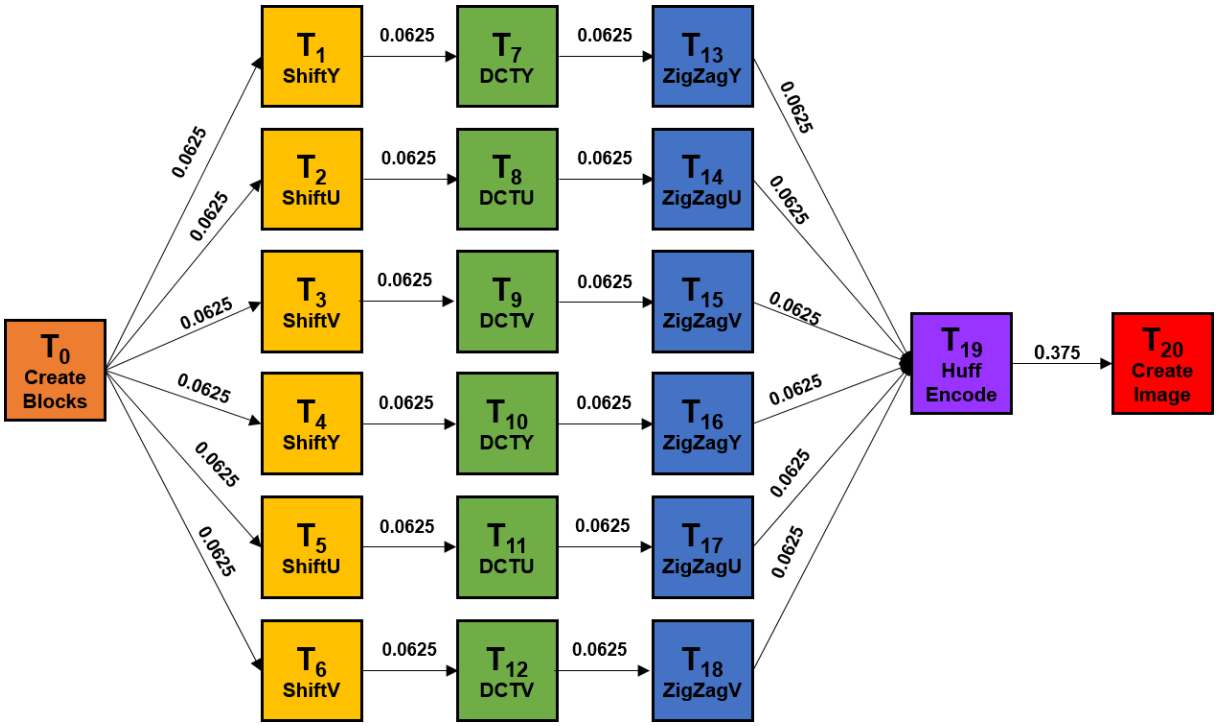
dostatak predloženog DAG modela aplikacije je u nemogućnosti prikaza ciklusa u aplikaciji. No, u većini aplikacija koje su namijenjene za ugradbene sustave ciklusi se pojavljuju najčešće samo na vršnoj razini tj. kompletan slijed operacija se ponavlja za svaki blok podataka (npr. AES) pa je moguće "razmotati" ciklus i prikazati ga slijedno ponavljanjem zadaća koje čine ciklus.

Model aplikacije se zapisuje u XML formatu. Vršni čvor je *application* koji predstavlja aplikaciju. Čvor *application* ima atribut *name* u kojem je zapisano ime aplikacije te može sadržavati neograničeni broj čvorova *task* koji predstavljaju procedure u aplikaciji (zadaće, engl. *task*). Čvor *task* ima atribute *id* i *name* kojima je označen redni broj čvora odnosno naziv procedure. Redni brojevi se dodjeljuju čvorovima tako što se čvorovi poredaju prema podatkovnoj međuovisnosti. Svaki čvor *task* osim početnog sadrži jedan ili više čvorova *pred*. Vrijednost čvora *pred* je redni broj (*id*) čvora prethodnika trenutnog čvora. Čvor *pred* ima jedan atribut *dataSize* u kojem je zapisana količina podataka koja se razmjenjuje između tog čvora i roditelja (*task*) u KB. U tablici 4.1 su sistematizirani svi elementi modela aplikacije.

Tablica 4.1: Elementi modela aplikacije

Element	Mogući pod-elementi	Atributi	
		Ime	*Opis*
application	task	name	*ime aplikacije*
task	pred	id	*redni broj čvora*
		name	*ime čvora*
pred	-	dataSize	*količina podataka u KB*

Na primjeru aplikacije JPEG za sliku veličine dva bloka izrađen je model aplikacije. Aplikacija JPEG se sastoji od procedura *CreateBlocks*, *Shift*, *DCT*, *ZigZag*, *HuffEncode* i *CreateImage*. Iz ulaznog skupa podataka operacija *CreateBlocks* stvara blokove veličine 8x8 piksela, a svaki piksel ima 3 komponente: Y, U i V. Nakon toga se za svaki blok i za svaku komponentu, neovisno o drugim komponentama, treba izvesti lanac procedura *Shift*, *DCT* i *ZigZag*. Nakon što su ti lanci gotovi za sve blokove, izvode se slijedno procedure *HuffEncode* i *CreateImage* nad svim podacima. Model aplikacije je prikazan u obliku grafa na slici 4.3 te XML zapisa na slici 4.4.



Slika 4.3: Primjer DAG-a aplikacije JPEG za sliku veličine dva bloka

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<application>
  <task id="0" name="T0-create_blocks"/>
  <task id="1" name="T1-shiftY">
    <pred dataSize="0.0625">0</pred>
  </task>
  <task id="2" name="T2-shiftU">
    <pred dataSize="0.0625">0</pred>
  </task>
  <task id="3" name="T3-shiftV">
    <pred dataSize="0.0625">0</pred>
  </task>
  <task id="4" name="T4-shiftY">
    <pred dataSize="0.0625">0</pred>
  </task>
  <task id="5" name="T5-shiftU">
    <pred dataSize="0.0625">0</pred>
  </task>
  <task id="6" name="T6-shiftV">
    <pred dataSize="0.0625">0</pred>
  </task>
  <task id="7" name="T7-DCTY">
    <pred dataSize="0.0625">1</pred>
  </task>
  <task id="8" name="T8-DCTU">
    <pred dataSize="0.0625">2</pred>
  </task>
  <task id="9" name="T9-DCTV">
    <pred dataSize="0.0625">3</pred>
  </task>
  <task id="10" name="T10-DCTY">
    <pred dataSize="0.0625">4</pred>
  </task>
  <task id="11" name="T11-DCTU">
    <pred dataSize="0.0625">5</pred>
  </task>
  <task id="12" name="T12-DCTV">
    <pred dataSize="0.0625">6</pred>
  </task>
  <task id="13" name="T13-ZZY">
    <pred dataSize="0.0625">7</pred>
  </task>
  <task id="14" name="T14-ZZU">
    <pred dataSize="0.0625">8</pred>
  </task>
  <task id="15" name="T15-ZZV">
    <pred dataSize="0.0625">9</pred>
  </task>
  <task id="16" name="T16-ZZY">
    <pred dataSize="0.0625">10</pred>
  </task>
  <task id="17" name="T17-ZZU">
    <pred dataSize="0.0625">11</pred>
  </task>
  <task id="18" name="T18-ZZV">
    <pred dataSize="0.0625">12</pred>
  </task>
  <task id="19" name="T19-HuffEncode">
    <pred dataSize="0.0625">13</pred>
    <pred dataSize="0.0625">14</pred>
    <pred dataSize="0.0625">15</pred>
    <pred dataSize="0.0625">16</pred>
    <pred dataSize="0.0625">17</pred>
    <pred dataSize="0.0625">18</pred>
  <task id="20" name="T20-createImage">
    <pred dataSize="0.375">19</pred>
  </task>
</application>
```

Slika 4.4: Primjera modela aplikacije JPEG za sliku veličine 2 bloka

4.2.2 Model platforme

Heterogena MPSoC platforma tipično se sastoji od desetak pa sve do stotinjak procesnih elemenata, nekoliko memorijskih elemenata te različitog namjenskog sklopovlja. Procesori su najčešće različitih vrsta, a mogu biti iz različitih porodica iste arhitekture (npr. ARM i Microblaze koji su po arhitekturi RISC procesori) ili potpuno različitih arhitektura (npr. RISC, DSP, GPU itd). Memorije su također heterogene, različitih vrsta i kapaciteta. Nužno je napomenuti da se priručna memorija (engl. *cache*) smatra sastavnim dijelom procesora te kada se u kontekstu MPSoC platforme govori o memorijama, misli se na memorijske elemente koji su fizički odvojeni od procesora. Procesori su s memorijama povezani različitim vrstama sabirnica, pri čemu ne mora svaki procesor biti povezan sa svakom memorijom.

U SLD pristupu, model aplikacije predstavlja opis željenog ponašanja, a model platforme je opis strukture tj. arhitekture sustava na kojem će se izvoditi ta aplikacija. Komponente platforme ostvaruju željeno ponašanje sustava tako što se na procesnim elementima izvode zadaće (tj. procedure u aplikaciji), a podaci se razmjenjuju putem memorija. Model platforme se sastoji od skupa parametriziranih komponenti (procesori, memorije) i sheme njihovog povezivanja. Komponente se parametriziraju na visokoj razini apstrakcije bez implementacijskih detalja. Za procesore se definira trajanje izvođenja pojedinih operacija (transakcija) ili čak cijelih procedura. Za memorije se najčešće navodi kapacitet, propusnost, broj priključnica (engl. *port*) i sl. Detalji vezani uz rad priručne memorije, operacijski sustav, protokole komunikacije i sl. se u ovim idealiziranim modelima izostavljaju. Model izgrađen na ovaj način slijedi SLD načelo razdvajanja komunikacije od izračuna te omogućava pridruživanje (engl. *mapping*) na način da se procedure pridružuju procesorima a komunikacijski kanali memorijama koje služe kao poveznica između dva procesora. Iako ovakav pristup smanjuje točnost procjene performansi sustava, on omogućava brzu i efikasnu pretragu prostora rješenja.

Parametrizirani modeli platforme visoke razine apstrakcije korišteni su i u ranije spomenutim alatima Daedalus i MAPS. Za reprezentaciju modela korišteni su grafovi i XML zapis. Posljedično, model arhitekture višeprocesorske platforme predložen u ovom radu se također prikazuje kao graf u kojem postoje dvije vrste čvorova: procesorski i memorijski. Procesorski čvorovi su povezani s memorijskim čvorovima i međusobno komuniciraju upisom podataka u memoriju odnosno čitanjem podataka iz memorije. Formalni model platforme glasi:

$$PLAT = (P, M, L). \tag{4.3}$$

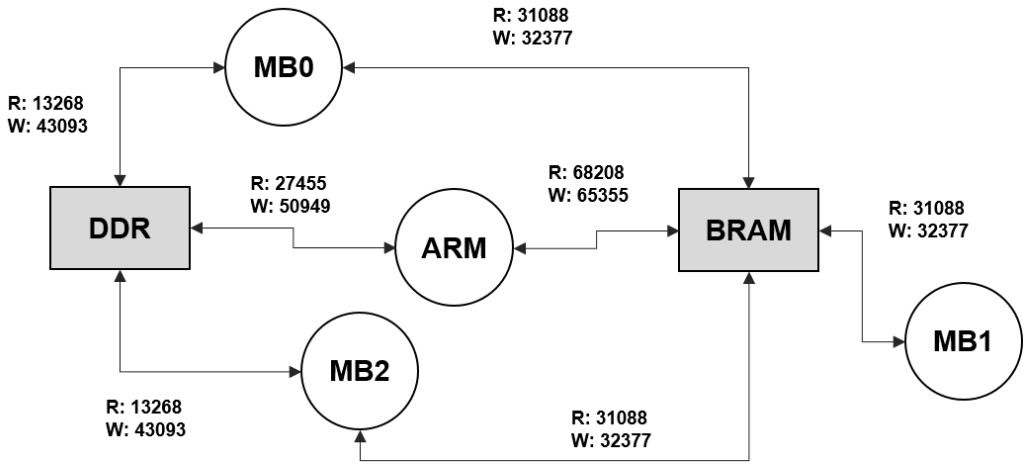
Čvorovi $P_i \in P$ predstavljaju procesore te je za svaki procesor definirano trajanje izvođenja pojedinih procedura koje su prikazane kao čvorovi u grafu modela aplikacije. Trajanje izvođenja pojedine procedure za pojedini procesor dobiva se pomoću metode za procjenu trajanja izvođenja na temelju elementarnih operacija opisane u prethodnom poglavlju. Pri tome je nužno imati dostupne profile aplikacije i platforme temeljene na elementarnim operacijama, kakvi su opisani u poglavlju 3.

Čvorovi $M_i \in M$ predstavljaju memorijske elemente. Za svaki memorijski element definirana je veličina te broj i vrsta priključnica (engl. *ports*) preko kojih se može čitati (R), pisati (W) ili oboje (RW).

Bridovi $Li \in L$ su veze između procesnih i memorijskih elemenata. Za svaku vezu definirana je brzina čitanja iz memorije i brzina upisa u memoriju. Čitanje iz memorije je modelirano kao poziv elementarne operacije MEM_BLOCK gdje je adresa podatka koji se dohvaća iz memorijskog elementa zadana kao izvor. Pisanje u memoriju je također modelirano kao poziv elementarne operacije MEM_BLOCK, ali je tada adresa podatka koji se pohranjuje u memorijski element zadana kao odredište. U modelu memorijskog elementa smatra se da sve priključnice imaju istu brzinu čitanja, odnosno pisanja.

U ovom modelu, memorije se promatraju isključivo u kontekstu komunikacije između procesnih elementa, a ne u smislu pohrane instrukcija i podataka koji se izvode na nekom procesoru. Memorija za pohranu instrukcija i podataka je za svaki procesor unaprijed zadana i nepromjenjiva i to je ona memorija koja je korištena kao dio konfiguracije procesor-memorija prilikom izrade profila platforme na temelju elementarnih operacija.

Na slici 4.5 je ilustriran primjer grafa modela platforme koja se sastoji od jednog ARM i tri Microblaze procesora koji razmjenjuju podatke putem BRAM i DDR memorija.



Slika 4.5: Primjer grafa modela platforme koja se sastoji od jednog ARM i tri Microblaze procesora koji razmjenjuju podatke putem BRAM i DDR memorija

Model platforme se zapisuje u XML formatu. Vršni čvor je *platform* koji predstavlja platformu. Čvor *platform* ima atribut *name* u kojem je zapisano ime aplikacije te atribut *specType* koji označava da li se radi o potpuno povezanoj platformi ili ne. Čvor *platform* sadrži čvorove *mem* koji predstavljaju memorije i čvorove *proc* koji predstavljaju procesore. Čvor *mem* ima atribute *id* - redni broj memorije, *name* - naziv memorije, *rPorts* - broj priključnica za čitanje, *wPorts* - broj priključnica za pisanje, *rwPorts* - broj priključnica za čitanje i pisanje te *size* veličina memorije. Čvorovi *proc* imaju atribute *id* i *name*. Oni sadrže čvorove *link* i *comp*. Čvorom *link* je definirana veza između procesora i memorije: vrijednost čvora *link* je *id* memorije. Čvor *link* ima atribute *rspeed* - brzina čitanja i *wspeed* - brzina pisanja u memoriju. Za svaku proceduru iz modela aplikacije postoji po jedan čvor *comp* čija je vrijednost duljina trajanja izvođenja te procedure na tom procesoru. Ako proceduru nije moguće izvesti na procesoru tada je vrijednost beskonačno. Kao što je ranije spomenuto, trajanje izvođenja procedure računa se na temelju profila elementarnih operacija. Čvor *comp* ima atribut *taskId* koji odgovara *id*-u te procedure u modelu aplikacije. U tablici 4.2 su navedeni svi elementi modela platforme.

Tablica 4.2: Elementi modela platforme

Element	Mogući pod-elementi	Atributi	
		Ime	*Opis*
platform	mem, proc	name	*ime procedure*
		specType	*povezanost elemenata platforme (0 - potpuna povezanost; 2 - nepotpuna povezanost)*
mem	-	id	*redni broj memorije*
		name	*ime čvora*
		rPorts	*broj priključnica za čitanje*
		wPorts	*broj priključnica za pisanje*
		rwPorts	*broj priključnica za čitanje i pisanje*
		size	*kapacitet memorije*
proc	link, comp	id	*redni broj procesora*
		name	*naziv procesora*
link	-	rspeed	*brzina čitanja iz memorije u KBps*
		wspeed	*brzina pisanja u memoriju u KBps*
comp	-	taskId	*id procedure u modelu aplikacije*

Za model prikazan na slici 4.5 dan je XML zapis na slici 4.6. Parametri pojedinih procesora (vrijednosti čvorova *comp*) se odnose na model aplikacije JPEG iz prethodne sekcije.

```
1  <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2  <platform specType="2">
3    <mem id="0" name="BRAM" rPorts="0" rwPorts="2" size="128.0" wPorts="0"
       />
4    <mem id="1" name="DDR" rPorts="0" rwPorts="1" size="1024.0" wPorts="0"
       />
5    <proc id="0" name="ARM">
6      <link rspeed="68208.0" wspeed="65355.0">0</link>
7      <link rspeed="27455.0" wspeed="50949.0">1</link>
8      <comp taskId="0">4.4499E-5</comp>
9      <comp taskId="1">5.1525E-7</comp>
10     <comp taskId="2">5.1525E-7</comp>
11     <comp taskId="3">5.1525E-7</comp>
12     <comp taskId="4">5.1525E-7</comp>
```

```
13      <comp taskId="5">5.1525E-7</comp>
14      <comp taskId="6">5.1525E-7</comp>
15      <comp taskId="7">3.9725E-6</comp>
16      <comp taskId="8">3.9725E-6</comp>
17      <comp taskId="9">3.9725E-6</comp>
18      <comp taskId="10">3.9725E-6</comp>
19      <comp taskId="11">3.9725E-6</comp>
20      <comp taskId="12">3.9725E-6</comp>
21      <comp taskId="13">4.3675E-7</comp>
22      <comp taskId="14">4.3675E-7</comp>
23      <comp taskId="15">4.3675E-7</comp>
24      <comp taskId="16">4.3675E-7</comp>
25      <comp taskId="17">4.3675E-7</comp>
26      <comp taskId="18">4.3675E-7</comp>
27       <comp taskId="19">7.35531E-6</comp>
28      <comp taskId="20">1.5108E-7</comp>
29    </proc>
30    <proc id="1" name="MB0">
31      <link rspeed="31088.0" wspeed="32377.0">0</link>
32      <link rspeed="13268.0" wspeed="43093.0">1</link>
33      <comp taskId="0">2.0582E-4</comp>
34      <comp taskId="1">2.853125E-6</comp>
35      <comp taskId="2">2.853125E-6</comp>
36      <comp taskId="3">2.853125E-6</comp>
37      <comp taskId="4">2.853125E-6</comp>
38      <comp taskId="5">2.853125E-6</comp>
39      <comp taskId="6">2.853125E-6</comp>
40      <comp taskId="7">5.5675E-5</comp>
41      <comp taskId="8">5.5675E-5</comp>
42      <comp taskId="9">5.5675E-5</comp>
43      <comp taskId="10">5.5675E-5</comp>
44      <comp taskId="11">5.5675E-5</comp>
45      <comp taskId="12">5.5675E-5</comp>
46      <comp taskId="13">3.613875E-6</comp>
47      <comp taskId="14">3.613875E-6</comp>
48      <comp taskId="15">3.613875E-6</comp>
49      <comp taskId="16">3.613875E-6</comp>
50      <comp taskId="17">3.613875E-6</comp>
51      <comp taskId="18">3.613875E-6</comp>
52       <comp taskId="19">0.002611E-4</comp>
53      <comp taskId="20">1.22185E-5</comp>
54    </proc>
55      <proc id="1" name="MB1">
56      <link rspeed="31088.0" wspeed="32377.0">0</link>
57      <comp taskId="0">2.0582E-4</comp>
58      <comp taskId="1">2.853125E-6</comp>
59    ...
```

```
60      <comp taskId="20">1.22185E-5</comp>
61    </proc>
62     <proc id="1" name="MB2">
63      <link rspeed="31088.0" wspeed="32377.0">0</link>
64      <link rspeed="13268.0" wspeed="43093.0">1</link>
65      <comp taskId="0">2.0582E-4</comp>
66      <comp taskId="1">2.853125E-6</comp>
67    ...
68      <comp taskId="20">1.22185E-5</comp>
69    </proc>
70  </platform>
```

Slika 4.6: Primjera modela platforme izgrađenog za aplikaciju JPEG za sliku veličine dva bloka

4.3 Metoda pretrage prostora oblikovanja

Metoda pretrage prostora oblikovanja predložena u ovom radu pokušava pronaći optimalno pridruživanje dijelova aplikacije na dijelove platforme pri čemu je kôd aplikacije zadan fiksno, a platforma labavo: zadana je vrsta elemenata koji se mogu koristiti i maksimalni broj instanci svake vrste elemenata. Cilj pretrage jest odabir elemenata od koji će biti izgrađena platforma tako da budu najpogodniji za izvođenje pojedinih dijelova aplikacije, tj. pronalaženje optimalne sheme pridruživanja dijelova aplikacije na elemente platforme (engl. *mapping scheme*) te određivanje redoslijeda izvođenja. Pri tome se procedure pridružuje procesorima, a komunikacijske kanale memorijama. Rezultat pretrage prostora oblikovanja je konačna fiksirana definicija platforme, shema pridruživanja dijelova aplikacije na elemente platforme te redoslijed njihovog izvođenja.

Kao što je opisano u potpoglavlju 4.1, velik dio metoda koje se bave pretragom prostora oblikovanja i raspoređivanjem često u obzir uzimaju samo raspoređivanje dijelova aplikacije na procesore, dok je memorijska konfiguracija unaprijed zadana i fiksna. Razlog tome je što razmatranje različitih memorijskih konfiguracija znatno povećava prostor mogućih rješenja i time usložnjava i produžuje trajanje pretrage. Međutim, uzimajući u obzir činjenicu da je vrijeme pristupa memoriji i do nekoliko redova veličine sporije od radnog takta procesora, iznimno je važno pri pretrazi prostora oblikovanja uzeti u obzir memorijsku konfiguraciju. U ovom radu su razmotrena i uspoređena dva pristupa:

1. *istovremeni* - procedure i komunikacijski kanali se istovremeno pridružuju procesorima odnosno na memorijama;
2. *slijedni* - najprije procedure pridružuju procesorima, a zatim komunikacijski kanali memorijama.

4.3.1 Definicija problema

Pretraga prostora oblikovanja se svodi na problem pronalaženja optimalnog pridruživanja dijelova aplikacije na dijelove platforme. Ako je aplikacija formalno zadana kao skup

$$APP = (T, Ch), \tag{4.4}$$

a platforma kao skup

$$PLAT = (P, M, L), \tag{4.5}$$

jedno pridruživanje je uređeni par

$$(\mu_t : T \to P, \mu_{Ch} : Ch \to M). \tag{4.6}$$

Pridruživanje komunikacijskih kanala između dvije procedure (t_i i t_j), koje su u odnosu prethodnik-sljedbenik (tj. podatkovno ovisne) može imati dva slučaja:

1. dvije slijedne zadaće su dodijeljene na isti procesor - komunikacijski kanal između njih se zanemaruje budući da u tom slučaju nije potrebno razmjenjivati podatke između tih procedura preko vanjske memorije;
2. dvije slijedne zadaće su dodijeljene na dva različita procesora - komunikacijski kanal se pridružuje jednom od dijeljenih memorijskih elemenata koji je povezan s oba procesora.

Pri razmatranju problema pridruživanja, MPSoC platforme imaju ograničenje da vrlo često nisu potpuno povezane tj. nisu svi procesori povezani sa svim memorijama. To znači da sva moguća pridruživanja nisu nužno i ispravna tj. izvediva. Stoga nužan uvjet da bi pridruživanje bilo ispravno jest da ako su dvije procedure koje su u odnosu prethodnik-sljedbenik dodijeljene na dva različita procesora, komunikacijski kanal između njih mora biti dodijeljen na memoriju koja je povezana s oba procesora i to mora vrijediti za sve procedure u aplikaciji.

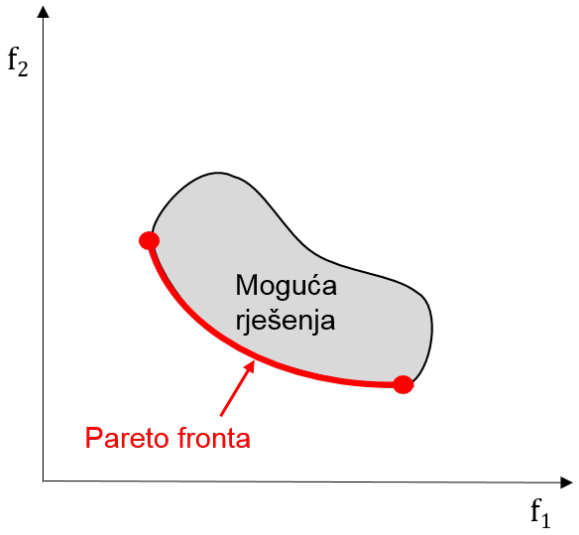
Pri odabiru između svih mogućih rješenja moguće je odrediti više kriterija prema kojima se određuje kvaliteta, koja se još i naziva dobrota rješenja (engl. *fitness*): vrijeme izvođenja, zauzeće resursa, broj elemenata platforme, potrošnja energije, cijena itd. Odabir rješenja prema više od jednog kriterija je problem *višekriterijske optimizacije* [60] i općenito se izražava kao

$$min\{f_1(x), f_2(x), ..., f_k(x)\};\ x \in S, k \geq 2. \tag{4.7}$$

Funkcije cilja (engl. *objective function*) $f_k : \mathbb{R}^n \to \mathbb{R}$ se minimiziraju istovremeno. Vektori odluke (engl. *decision vectors*) $x = (x_1, x_2, ..., x_n)^T$ pripadaju nepraznoj (engl. *non-empty*) regiji mogućih rješenja $S \subset \mathbb{R}^n$. Vektori cilja (engl. *objective vectors*) su slike vektora odluke i sastoje se od vrijednosti funkcija cilja: $f(x) = (f_1(x), f_2(x), ..., f_k(x))^T$. U predloženoj metodi vektor odluke čine: pridruživanje procedura na procesore i pridruživanje komunikacijskih kanala na memorije. Za funkcije cilja su odabrana dva kriterija: ukupno vrijeme izvođenja i ukupan broj korištenih elemenata na platformi (procesora i memorija). Cilj je istovremeno minimizirati obje funkcije. Ukupno vrijeme izvođenja se dobije izradom statičkog rasporeda izvođenja bez prekida pri čemu je nužno poštovati odnose među procedurama kako je zadano u

modelu aplikacije.

U višekriterijskoj optimizaciji za netrivijalne (engl. *non-trivial*) probleme ne postoji jedno rješenje koje minimizira sve kriterije istovremeno. Umjesto toga postoje *Pareto optimalna rješenja* - rješenja koja se ne mogu poboljšati po niti jednom kriteriju a da se ne pogoršaju po nekom drugom. Takva rješenja se još nazivaju i *nedominirana* (engl. *non-dominated*). Formalno se definira da je rješenje $x' \in S$ Pareto optimalno ako ne postoji neki drugi $x \in S$ takav da $f_i(x) \leq f_i(x')$ za sve $i = 1, ..., k$ i $f_j(x) \leq f_j(x')$ za barem jedan indeks j. Sva Pareto optimalna rješenja čine tzv. *Pareto frontu* kao što je prikazano na slici 4.7. Sva Pareto optimalna rješenja su u matematičkom smislu jednako dobra te se konačna odluka o odabiru jednog rješenja najčešće donosi izvana na temelju ekspertnih znanja.



Slika 4.7: Pareto fronta

4.3.2 Evolucijska višekriterijska optimizacija

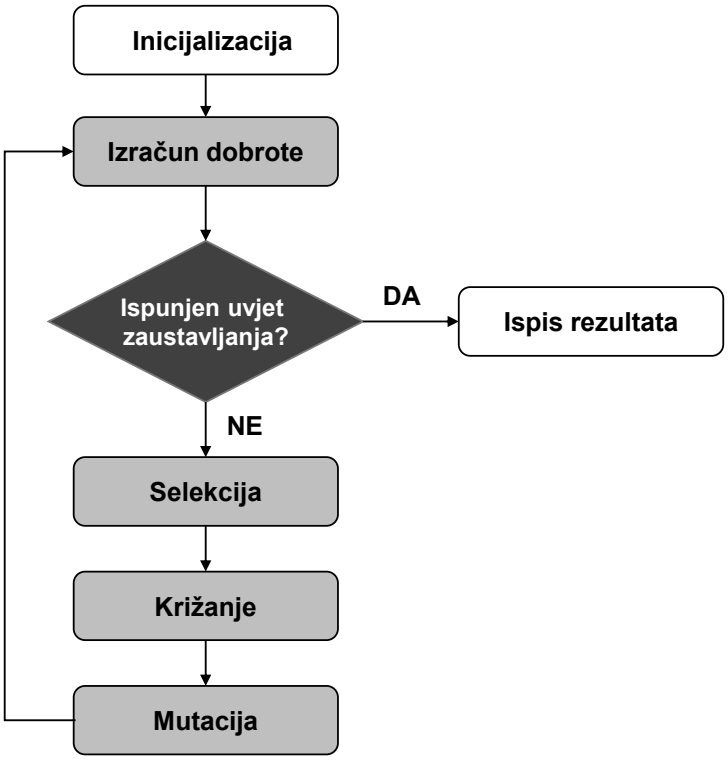
Evolucijski algoritmi za višekriterijsku optimizaciju (engl. *Multiobjective Evolutionary Algorithms* - MOEA) su općeprihvaćene metode za rješavanje NP-teških problema. U tu kategoriju također pripadaju problemi pridruživanja i raspoređivanja u heterogenim MPSoC sustavima pa stoga većina radova iz područja koristi metode evolucijskog računarstva koje prilagođava konkretnom problemu. Najzastupljeniji višekriterijski optimizacijski algoritmi su SPEA2, korišten u [28, 46], i NSGA-II, korišten u [26, 46]. Metoda opisana u ovom radu će se također temeljiti na genetskom algoritmu NSGA-II, opisanom u [12].

Glavne karakteristike genetskih algoritama (GA) su: rad nad populacijom, lokalna pretraga

i iterativno poboljšavanje rješenja. Populacija je skup jedinki, a veličina populacije je konstantna. Svaka jedinka (tj. vektor odluke) predstavlja jedno rješenje problema. Jedinke se još nazivaju i *kromosomi*. Kromosom se sastoji od *gena* koji predstavljaju dijelove rješenja, npr. procedure u aplikaciji koje se pridružuju na neki procesor. Struktura i način zapisa kromosoma se prilagođava problemu. Svaka jedinka je karakterizirana svojom dobrotom - vrijednosti funkcije cilja.

Genetski algoritmi rade na principu *lokalne pretrage* - pretražuje se susjedstvo trenutnog rješenja u potrazi za boljim rješenjem od trenutnog. Susjedno rješenje se definira kao rješenje do kojeg se može doći iz trenutog koristeći dobro definiranu modifikaciju. U slučaju GA to su operatori *križanja* (engl. *crossover*) i *mutacije* (engl. *mutation*). Križanje je postupak stvaranja novog rješenja rekombinacijom gena dvaju postojećih rješenja, što znači da se u GA koncept susjedstva ne temelji samo na jednoj jedinci (rješenju). Postoje različite tehnike križanja [60]: križanje u jednoj točki (engl. *one-point crossover*), križanje u dvije točke (engl. *two-point crossover*), uniformno križanje (engl. *uniform crossover*) itd. Mutacija je nasumična promjena jednog ili više gena na jedinki nastaloj križanjem. Križanje i mutacija osiguravaju diversifikaciju populacije. Ovakvo direktno pretraživanje susjedstva je znatno jednostavnije za implementaciju i lakše prilagodljivo širokom spektru optimizacijskih problema za razliku od postupaka koji koriste informacije o gradijentu prilikom pretrage, kao npr. cjelobrojno programiranje [61].

Genetski algoritmi su primjer poboljšavajuće (engl. *improvement*) heuristike koja odmah u prvom koraku kreće od nasumično odabranog cjelovitog rješenja i iterativno ga poboljšava mijenjajući njegove dijelove. GA ne garantiraju pronalazak optimalnog rješenja, ali konvergiraju prema optimalnom rješenju zadanog problema. Svaka iteracija se naziva generacija. U svakoj generaciji se odabiru najbolje jedinke za reprodukciju - *selekcija*. Postoje različite metode selekcije: rangiranje, turnirska selekcija itd. Iz odabranih jedinki se križanjem i mutacijom stvaraju nova rješenja koja prelaze u sljedeću generaciju, a stara generacija se odbacuje. Većina postojećih genetskih algoritama koristi i načelo *elitizma* pri odabiru jedinki koje ulaze u sljedeću generaciju. To znači da najbolja rješenja iz stare populacije nepromijenjena prelaze u novu generaciju i na taj način se osigurava monotono nedegradirajuća kvaliteta rješenja [60]. Iteracije se ponavljaju sve dok se ne zadovolji kriterij zaustavljanja što može biti: broj iteracija, vrijeme izvođenja, dobrota rješenja itd. Tijek izvođenja GA je ilustriran na slici 4.8.



Slika 4.8: Tijek izvođenja genetskog algoritma

Višekriterijska optimizacija temeljena na genetskim algoritmima je aposteriori metoda koja radi u dva koraka. U prvom koraku se izvodi genetski algoritam kroz niz generacija kako bi se dobio skup nedominiranih točki koje su što bliže Pareto optimalnoj fronti. U drugom koraku se odabire jedna od točaka na temelju ekspertnog znanja i proglašava konačnim rješenjem. Budući da je evolucijska višekriterijska optimizacija heuristička metoda, ona ne garantira pronalaženje Pareto optimalnih rješenja (kao npr. kod cjelobrojnog programiranja). No, pomoću operatora varijacije i selekcije skup rješenja konstantno evoluira i poboljšava se te nakon konačnog broja iteracija dolazi vrlo blizu optimalne Pareto fronte.

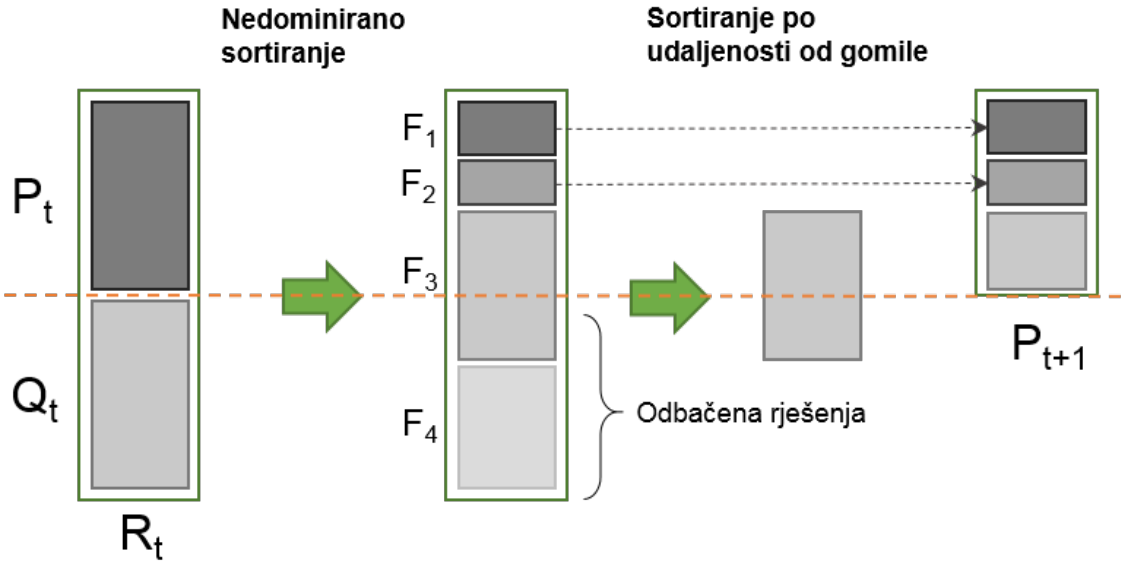
4.3.3 Prilagodba algoritma NSGA-II

Algoritam NSGA-II (puni naziv: *Elitist Non-dominated Sorting Genetic Algorithm*) [12] je jedna od najpopularnijih evolucijskih metoda koja nastoji pronaći Pareto optimalna rješenja i ima sljedeće tri karakteristike:

1. koristi elitizam,

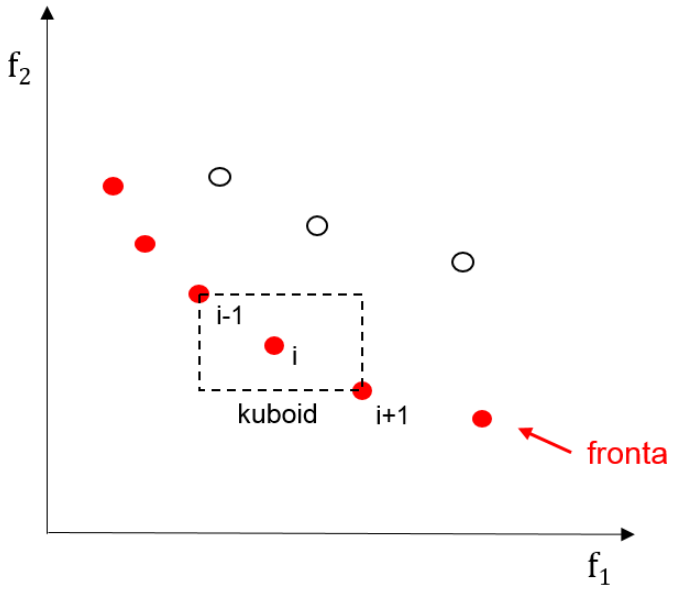
2. koristi mehanizam eksplicitnog očuvanja raznolikosti rješenja,
3. naglašava nedominirana rješenja.

Shematski prikaz algoritma NSGA-II se nalazi na slici 4.9. U svakoj generaciji t, populacija potomaka Q_t se stvara iz populacije roditelja P_t pomoću genetskih operatora križanja i mutacije. Jedinke za križanje se odabiru klasičnom 2-turnirskom selekcijom ($k = 2$), a kriterij selekcije je udaljenost od gomile. Nakon toga te dvije populacije se spajaju i tvore novu populaciju R_t veličine $2N$. Zatim se populacija R_t dijeli u fronte te se populacija koja se prenosi u sljedeću generaciju P_{t+1} puni točkama iz fronti populacije R_t, jedna po jedna. Punjenje počinje sa prvom nedominiranom frontom, zatim drugom itd. Budući da je populacija populacija R_t duplo veća od populacije P_{t+1} ne mogu sve točke prijeći u novu populaciju i točke koje nisu stale se brišu.



Slika 4.9: Shematski prikaz algoritma NSGA-II

Pri prebacivanju fronti iz R_t u P_{t+1}, moguće je da za zadnju frontu postoji manje slobodnih mjesta nego točaka u fronti. Tada se rješenja koja ulaze u sljedeću generaciju, umjesto nasumično, biraju prema raznolikosti. Radi se sortiranje točaka zadnje fronte po *udaljenosti od gomile* (engl. *crowding distance sorting*) i to u padajućem redoslijedu. Točke s vrha sortirane liste (najveća udaljenost) prelaze u sljedeću generaciju, a ostale se odbacuju. Udaljenost od gomile d_i za točku i se definira kao mjera prostora rješenja oko i koji nije okupiran s nekim drugim rješenjem iz populacije. Mjera d_i se računa procjenjujući granice kuboida koji se formira uzimajući u obzir najbliže susjede u prostoru rješenja kao vrhove, kao što je prikazano na slici 4.10.



Slika 4.10: Udaljenost od gomile

Kao osnova za implementaciju korištena je implementacija NSGA-II algoritma u radnom okviru MOEA [62]. To je knjižnica otvorenog kôda (LGPL v3) napisana u jeziku Java za razvoj optimizacijskih algoritama. U njoj postoji osnovna implementacija mnogih genetskih algoritama, između ostalih i NSGA-II, a radni okvir omogućava proširenja i modifikacije prema potrebi.

Osmišljene su dvije inačice pretrage prostora oblikovanja:

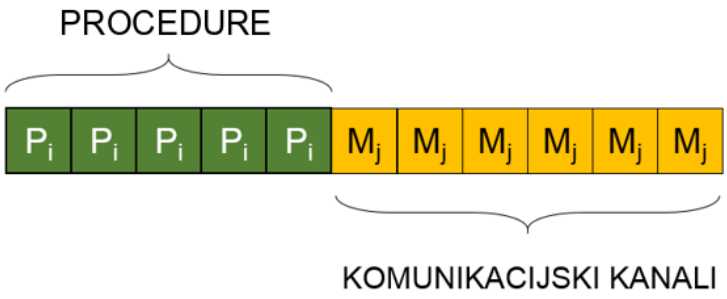
1. istovremena pretraga pod nazivom *Simultaneous Design Space Exploration* (SDSE),
2. dvostupanjska pretraga pod nazivom *Two-Step Design Space Exploration* (2SDSE)[a].

Istovremena pretraga - algoritam SDSE

Algoritam SDSE tijekom pretrage prostora oblikovanja istovremeno pridružuje procedure procesorima i komunikacijske kanale memorijama. Postupak započinje učitavanjem modela aplikacije i platforme. Na temelju toga se stvara populacija od 100 jedinki koje predstavljaju nasumično odabrana rješenja. Kromosom svake jedinke se sastoji od dvije vrste gena: onih koji predstavljaju procedure i onih koji predstavljaju komunikacijske kanale. Ako u modelu aplikacije ima M procedura i N komunikacijskih kanala, kromosom će ukupno imati $M+N$ gena te će prvih M gena u kromosomu predstavljati procedure u aplikaciji, a sljedećih N gena komunika-

[a]Izvorni kôd algoritama SDSE i 2SDSE je dostupan na adresi `https://gitlab.com/Frid/dsexplorer/tree/master/SDSE`.

cijske kanale. Redoslijed procedura i komunikacijskih kanala je zadan prema njihovom rednom broju u modelu aplikacije - vrijednost atributa *id*. Svaki gen je zapisan kao jedan broj u formatu pomičnog zareza u rasponu od 0 do 1. Taj broj se dekodira (engl. *gene decoding*) u redni broj procesora za gene koji predstavljaju procedure, odnosno u redni broj memorije za gene koji predstavljaju komunikacijske kanale. Redni broj procesora i memorija odgovara atributu *id* iz modela platforme. Na slici 4.11 je prikazana prethodno opisana struktura kromosoma.



Slika 4.11: Izgled kromosoma za algoritam SDSE

S obzirom da na tipičnoj MPSoC platformi ne moraju nužno svi procesori moći izvesti sve procedure, pri dekodiranju gena koji se odnose na procedure, vodilo se računa da se najprije odredi skup procesora koji mogu izvesti konkretnu proceduru i onda se dekodiralo uzmajući u obzir samo taj skup procesora. Na taj način izbjegnuta su nemoguća rješenja. U slučaju potpuno povezane platforme ne postoji problem s nemogućim rješenjima kad se radi o pridruživanju komunikacijskih kanala na memorije, ali to je stvarni problem kod rijetko povezanih platformi. Budući da u slučajevima nepotpuno povezanih platformi može doći do situacije da je broj nemogućih rješenja za nekoliko redova veličine veći od broja mogućih rješenja, ovaj problem nije moguće riješiti na jednostavan način o čemu će biti više riječi kasnije.

Nakon stvaranja početne populacije, stvara se potomstvo pomoću operatora križanja i mutacije. Za križanje i mutaciju korišteni su sljedeći operatori iz MOEA radnog okvira: *1x*, *2x*, *ux*, *sbx*, *spx*, *pcx*, *undx*, *pm* i *um*. Radi se o standardnim operatorima koji se koriste u genetskim algoritmima, a njihova implementacija u MOEA radnom okviru je opisana u [63].

Jedinke, tj. potencijalna rješenja, se evaluiraju prema dva kriterija: ukupnom trajanju izvođenja i broju korištenih procesora i memorija. Cilj je minimizirati oba kriterija. Izračun broja korištenih elemenata na platformi je trivijalan, no da bi se izračunalo ukupno trajanje izvođenja potrebno je minimalno izraditi statički raspored izvođenja procedura na temelju pridruživanja

zadanog u kromosomu uz poštivanje podatkovnih međuovisnosti iz modela aplikacije.

Izrada rasporeda odvija se na način kako je navedeno u algoritmima čiji je pseudokôd dan na slikama 4.12 i 4.13. Procedure se raspoređuju jedna po jedna, redoslijedom kako je zadano u modelu aplikacije. Za svaku proceduru se najprije iz kromosoma dekodira procesor na kojem se izvodi. Zatim se određuje raspored, tj. konkretni trenuci početka i završetka izvođenja procedure pozivom funkcije *rasporediProceduru*.

Algorithm 2 Izračun trajanja izvođenja

```
1:  for all procedure iz modela aplikacije do
2:      procesor = dekodiraj iz kromosoma procesor na kojem se izvodi procedura
3:      shift = 0.0                                   ▷ pomak u vremenu početka izvođenja
4:      procSlot = RASPOREDIPROCEDURU(procedura, procesor, reservSlots, shift)
5:      sortiraj reservSlots
6:      while true do
7:          if početak prve rezervacije u reservSlots != početak procSlot then
8:              obriši sve rezervacije
9:              shift = dohvati vrijeme početka procSlot
10:             RASPOREDIPROCEDURU(procedura, procesor, reservSlots, shift)
11:         else
12:             dodaj procSlot u listu zauzeća za procesor
13:             izađi iz petlje
14:         end if
15:     end while
16: end for
17: trajanjeIzvodenja = pronađi vrijeme završetka zadnjeg procesora
```

Slika 4.12: Pseudokôd postupka izračuna trajanja izvođenja

U apstraktnom modelu sustava, svaka *procedura* se izvodi u tri faze:

1. dohvat podataka od procedura koje joj prethode,
2. izračun,
3. slanje podataka procedurama koje joj slijede.

Prethodnici i sljedbenici se određuju prema odnosima u grafu modela aplikacije. Dohvaćanje podataka se odvija redom kako je svaki prethodnik gotov sa izvođenjem i podrazumijeva čitanje iz memorije na koju je dodijeljen komunikacijski kanal sa prethodnikom. Slanje podataka sljedbenicima se odvija redom kako su zadani komunikacijski kanali i podrazumijeva pisanje u memoriju na koju je dodijeljen komunikacijski kanal sa sljedbenikom. Ukupno trajanje izvođenja procedure jest vrijeme od trenutka početka čitanja iz memorije koja ju povezuje s prvim prethodnikom do trenutka završetka upisa u memoriju koja ju povezuje sa zadnjim sljedbenikom.

U skladu s time i raspoređivanje pojedine procedure ide u tri faze na način kako je prikazano u pseudokôdu funkcije *rasporediProceduru* danom na slici 4.13. Najprije se odrede svi prethodnici - varijabla *preds* u pseudokôdu i sortiraju rastuće prema redoslijedu završetka svakog prethodnika; najranije moguće vrijeme početka čitanja podataka od prvog prethodnika - *start_t* jest vrijeme završetka njegovog izvođenja. Zatim se za svaki *prethodnik* iz skupa *preds* iz kromosoma dekodira memorija na koju je dodijeljen komunikacijski kanal koji povezuje *proceduru* i taj *prethodnik*. Memorija se dekodira s obzirom na vrijednost odgovarajućeg gena, uzimajući u obzir samo memorije koje povezuju procesore na kojima su procedura i prethodnik. Na temelju količine podataka koje *procedura* dohvaća od *prethodnika* i brzine čitanja memorije koja ih povezuje određuje se trajanje komunikacije - *trajanjeKom*. Čitanje od sljedećeg prethodnika ne može početi prije nego što je završilo trenutno - *comm_t1*. Na kraju prve faze, ukupno trajanje dohvaćanja podataka od svih prethodnika je *comm_t1*.

Nakon toga se procedura izvodi na procesoru kojem je dodijeljena, što je označeno kao *exec_t*. Konačno, procedura šalje rezultate svog izvođenja svim procedurama koje joj slijede, a s kojima je povezana u modelu aplikacije - varijabla *succs* u pseudokôdu. Slanje podataka se odvija redom kako su zadani komunikacijski kanali i podrazumijeva pisanje u memoriju na koju je dodijeljen određeni komunikacijski kanal. Memorija se dekodira s obzirom na vrijednost odgovarajućeg gena, uzimajući u obzir samo memorije koje povezuju procesore na kojima su procedura i sljedbenik. Na temelju količine podataka koje *procedura* dohvaća od *sljedbenika* i brzine pisanja memorije koja ih povezuje određuje se trajanje komunikacije - varijabla *trajanjeKom*. Ukupno trajanje slanja podataka svim sljedbenicima je *comm_t2*. Ukupno vrijeme koje procedura zauzima na procesoru jest vrijeme od *start_t* do *comm_t2*.

Pristupi memoriji i procesoru se prate pomoću *vremenskih odsječaka* (engl. *slot*) koji predstavljaju vrijeme tijekom kojeg procedura pristupa memoriji iz koje čita ili u koju piše ili se izvodi na nekom procesoru. Za svaki procesor i za svaku memoriju definirana je sortirana *lista zauzeća* u kojoj su poredani zauzeti vremenski odsječci na tom procesoru ili memoriji. Za memorije postoji zasebna lista zauzeća za svaki priključak (engl. *port*) te se vodi računa radi li se o priključku preko kojeg se može čitati, pisati ili oboje. Kada npr. neka procedura želi dohvatiti podatke od jednog svog prethodnika to podrazumijeva dodavanje novog vremenskog odsječka u listi zauzeća na priključku za čitanje memorije na koju je dodijeljen komunikacijski kanal između te procedure i tog konkretnog prethodnika.

Da bi se dodao novi vremenski odsječak u listu nužno je znati vrijeme njegovog početka i

Algorithm 3 Raspoređivanje procedure

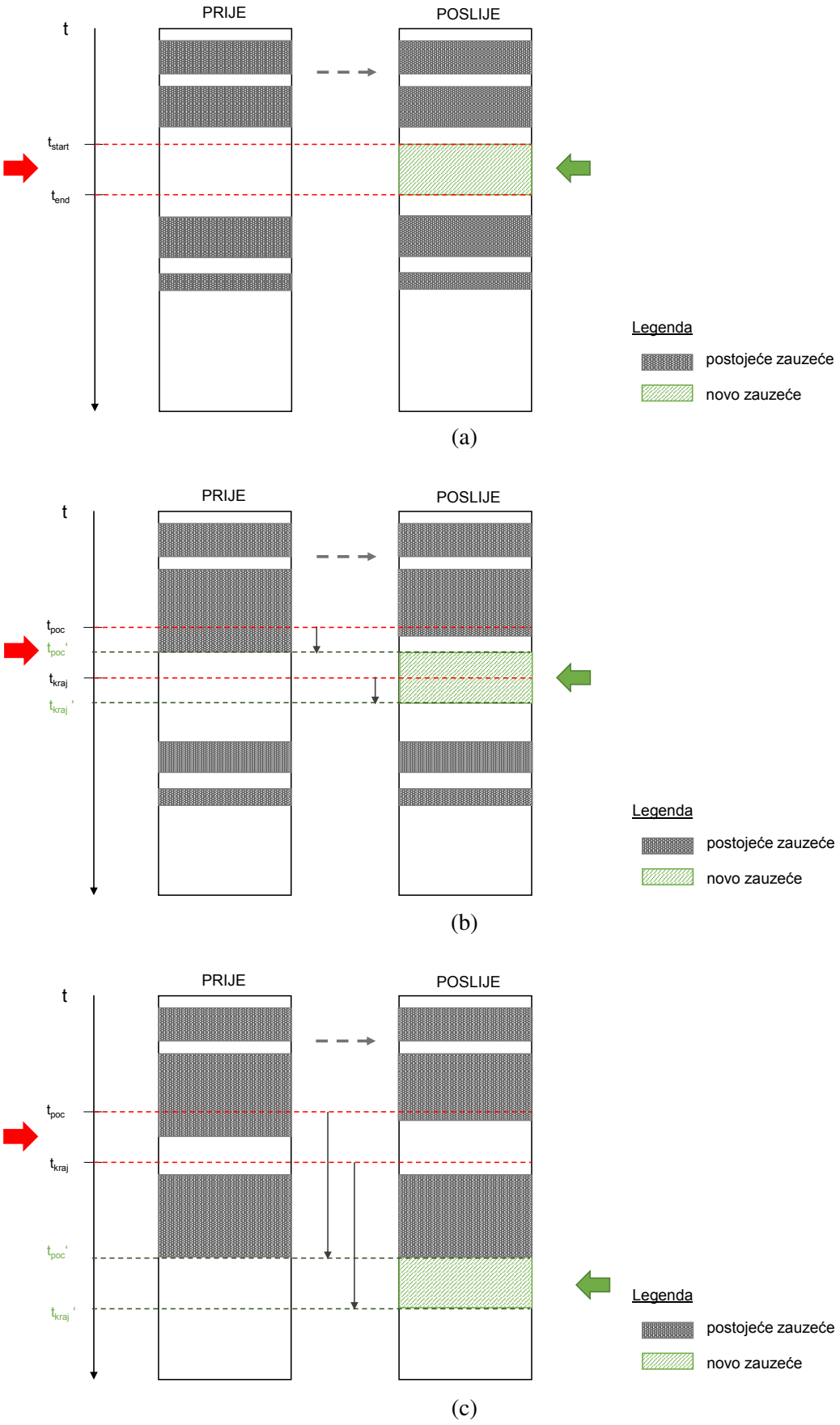
```
1:  function RASPOREDIPROCEDURU(procedura, procesor, reservSlots, shift)
2:      preds = svi prethodnici od procedura
3:      sortiraj preds po vremenima završetka izvođenja svakog prethodnika
4:      if shift == 0.0 then
5:          start_t = vrijeme završetka prvog prethodnika iz preds
6:      else
7:          start_t = shift                    ▷ vrijeme početka se pomiče zbog zauzeća procesora
8:      end if
9:      comm_t1 = start_t
10:     for each prethodnik ∈ preds do
11:         if prethodnik i procedura se ne izvode na istom procesoru then
12:             predMem = dekodiraj memoriju iz gena
13:             if vrijeme završetka prethodnika > comm_t1 then
14:                 comm_t1 = vrijeme završetka prethodnika
15:             end if
16:             trajanjeKom = kolPodataka(procedura, prethodnik)/brzinaČitanja(predMem)
17:             CS = pronađi vremenski odsječak u predMem za parametre comm_t1 i trajanjeKom
18:             comm_t1 = vrijeme završetka CS
19:             dodaj CS u listu zauzeća od predMem
20:             dodaj CS u reservSlots
21:         end if
22:     end for
23:     exec_t = trajanje izvođenja procedura na procesor
24:     comm_t2 = exec_t + comm_t1;
25:     succs = svi sljedbenici od procedura
26:     for each sljedbenik ∈ succs do
27:         succ_proc = procesor na kojem se izvodi sljedbenik
28:         succ_chan = komunikacijski kanal između procedura i sljedbenik
29:         if sljedbenik i procedura se ne izvode na istom procesoru then
30:             succMem = dekodiraj memoriju iz gena
31:             trajanjeKom = kolPodataka(sljedbenik, procedura)/brzinaPisanja(succMem
32:             CS = pronađi vremenski odsječak u sljedMem za parametre comm_t2 i trajanjeKom
33:             comm_t2 = vrijeme završetka CS
34:             dodaj CS u listu zauzeća od succMem ;
35:             dodaj CS u reservSlots
36:         end if
37:     end for
38:     PS = pronađi vremenski odsječak na procesoru za vrijeme od start_t do comm_t2
39:     return PS;
40: end function
```

Slika 4.13: Pseudokôd postupka raspoređivanja procedure

trajanje. Za memorije se trajanje odsječka izračunava na temelju podataka iz modela aplikacije i platforme, tj. količine podataka koja se razmjenjuje i brzine čitanja/pisanja u tu memoriju - linije 15 i 30 u algoritmu na slici 4.13. Za procesore je to ukupno vrijeme koje procedura provede na procesoru. Vrijeme početka se određuje tako da se na temelju ovisnosti iz modela aplikacije i do tada izrađenog rasporeda izračuna *trenutak najranijeg mogućeg početka* - linije 9, 14, 24 u algoritmu na slici 4.13. Zatim se u listi zauzeća traži vremenski razmak u trajanju većem ili jednakom trajanju vremenskog odsječka, a cilj je stvoriti novi vremenski odsječak sa vremenom

početka što bližem najranijem mogućem. Tu su moguća tri slučaja ilustrirana na slici 4.14. U prvom slučaju - slika (a), memorija ili procesor su slobodni u najranijem mogućem vremenu početka i to najmanje onoliko koliko traje odsječak. U drugom slučaju - slika (b), procesor ili memorija su zauzeti u zadanom vremenu početka, ali ako se vrijeme početka malo pomakne unaprijed postoji dovoljno velik vremenski razmak u koji se može smjestiti novi odsječak. U trećem slučaju - slika (c), na procesoru ili memoriji ne postoji nigdje dovoljno velik vremenski razmak nego se novi odsječak dodaje na kraj liste. Dodatno, za memorije kod kojih postoji više priključaka iste vrste paralelno se pokušava naći mjesto u listama zauzeća za sve priključke i vremenski odsječak se dodaje u onu listu u kojoj može imati najranije vrijeme početka.

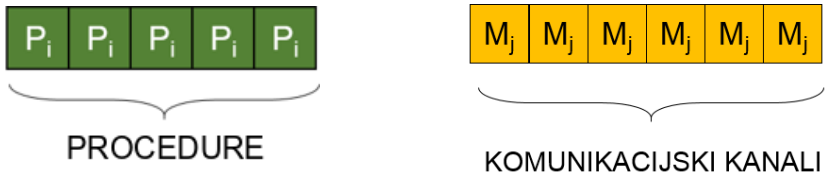
Budući da se najprije gleda zauzeće memorija, mogući je slučaj da procesor nije slobodan u trenutku kad treba započeti prva komunikacija s memorijom. Tada se pomiče vrijeme početka prve komunikacije na trenutak kada je procesor slobodan - varijabla $shift\_t$ i postupak raspoređivanja se ponavlja ispočetka i tako sve dok se vremena ne usklade kao što je prikazano u pseudokôdu na slici 3.12 - petlja *while*. Naravno, ovo ne vrijedi za prvu proceduru jer su tada sve liste zauzeća prazne. Također za prvu proceduru vrijeme *comm_t1* je *0.0* budući da nema prethodnika.



Slika 4.14: Ilustracija mogućih slučajeva pri dodavanju novog vremenskog odsječka u listu zauzeća

Dvostupanjska pretraga - algoritam 2SDSE

Algoritam 2SDSE najprije pridružuje procedure procesorima, a zatim komunikacijske kanale memorijama. Algoritam se odvija u dvije iteracije. U prvoj iteraciji se procedure pridružuju procesorima. Kromosom svake jedinke se sastoji samo od gena koji predstavljaju procedure pa ako u modelu aplikacije ima M procedura kromosom će ukupno imati M gena, kao što je prikazano na slici 4.15a. Pri dekodiranju gena također se vodi računa da se najprije odredi skup procesora koji mogu izvesti konkretnu proceduru i onda se dekodira uzmajući u obzir samo taj skup procesora. Na taj način izbjegnuta su nemoguća rješenja. Ostali parametri vezani uz format zapisa gena u kromosomu, broja jedinki, parametara križanja i mutacije su isti kao i u algoritmu SDSE. Jedinke, tj. potencijalna rješenja, se ponovno evaluiraju prema dva kriterija: ukupnom trajanju izvođenja i broju korištenih procesora te je cilj minimizirati oba kriterija.



(a) Kromosom u prvoj iteraciji - geni predstavljaju procedure

(b) Kromosom u drugoj iteraciji - geni predstavljaju komunikacijske kanale

Slika 4.15: Kromosomi u algoritmu 2SDSE

U prvoj iteraciji izrada rasporeda se odvija na vrlo sličan način kako je navedeno u pseudokôdu prikazanom na slikama 4.12 i 4.13 s time da se vrijeme komunikacije između procedura zanemaruje te je ukupno vrijeme izvođenja jedne procedure jednako *exec_t*. U skladu s time vremenski odsječci u trajanju *exec_t* se zauzimaju samo na procesoru. Pseudokôd izračuna trajanja izvođenja u prvoj iteraciji je prikazan na slici 4.16

Algorithm 4 Izračun trajanja izvođenja - iteracija 1

```
1:  for all procedure iz modela aplikacije do
2:      procesor = dekodiraj iz kromosoma procesor na kojem se izvodi procedura
3:      preds = svi prethodnici od procedura
4:      sortiraj preds po vremenima završetka izvođenja svakog prethodnika
5:      start_t = vrijeme završetka zadnjeg prethodnika iz preds
6:      exec_t = trajanje izvodenja procedura na procesor
7:      PS = pronadi vremenski odsječak na procesoru s početkom u start_t i trajanjem exec_t
8:      dodaj PS u listu zauzeća za procesor
9:  end for
10: trajanjeIzvodenja = vrijeme završetka zadnjeg procesora
```

Slika 4.16: Pseudokôd postupka izračuna trajanja izvođenja - iteracija 1

U drugoj iteraciji algoritma komunikacijski se kanali pridružuju memorijama. Kromosom svake jedinke se sastoji samo od gena koji predstavljaju komunikacijske kanale pa ako u modelu aplikacije ima N procedura kromosom će ukupno imati N gena, kao što je prikazano na slici 4.15b. Memorija se dekodira s obzirom na vrijednost odgovarajućeg gena, uzimajući u obzir samo memorije koje povezuju procesore na kojima su procedura i prethodnik. Ostali parametri vezani uz format zapisa gena u kromosomu, broja jedinki, parametara križanja i mutacije su isti kao i u prvoj inačici algoritma. Jedinke, tj. potencijalna rješenja, se ponovno evaluiraju prema dva kriterija: ukupnom trajanju izvođenja te broju korištenih procesora i memorija, a cilj je minimizirati oba kriterija. U ovoj iteraciji je postupak raspoređivanja identičan onome iz algoritma SDSE - slike 4.12 i 4.13 s time što nije potrebno raditi dekodiranje procesora budući da su oni fiksno zadani na kraju prve iteracije.

4.4 Evaluacija

Evaluacija je napravljena na umjetno stvorenim modelima i modelima stvarnih aplikacija i platforma[b]. Umjetni modeli aplikacija i platforma stvoreni su pomoću biblioteke *PSPLib*[c] [64] te omogućuju jednostavno variranje parametara konfiguracije kao što je omjer komunikacije i izračuna, ukupan broj te vrsta procesora i memorija, određivanje da li se neka aplikacija može izvesti na samo jednoj vrsti procesora ili na više njih i sl. Na taj način moguće je ispitati predloženu metodu u raznovrsnijem okruženju nego kada bi se za ispitivanje koristile samo stvarne

[b] Modeli su dostupni na adresi: https://gitlab.com/Frid/dsemodels.git
[c] Dostupno na: http://www.om-db.wi.tum.de/psplib/data.html

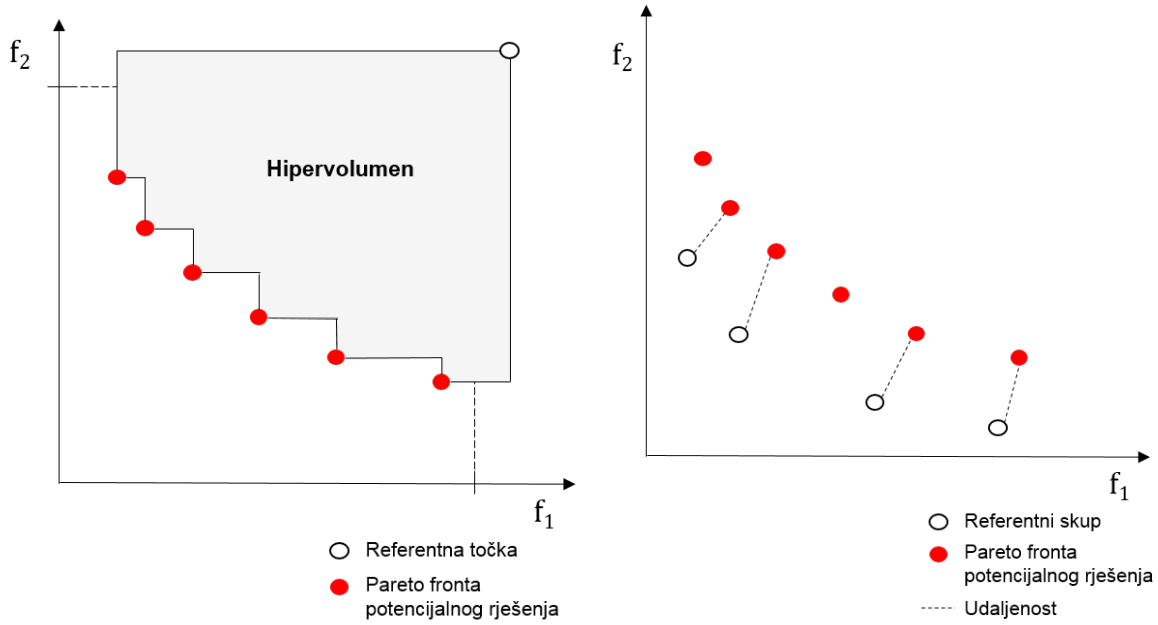
aplikacije i platforme. Odabrane stvarne aplikacije su JPEG kompresija i praćenje zrake (engl. *ray tracing* - RT) te platforme Xilinx ZC706 i Adapteva Parallella.

Budući da metode optimizacije temeljene na genetskim algoritmima ne jamče pronalazak optimalnog rješenja, evaluacija je napravljena međusobnom usporedbom rješenja dobivenih algoritmima SDSE i 2SDSE. U literaturi postoji mnoštvo mjera pomoću kojih je moguće uspoređivati rješenja dobivena različitim metodama [65], no s obzirom da sve optimizacijske metode nisu podjednako učinkovite za sve vrste problema [66] preporučuje se uvijek koristiti barem onoliki broj mjera koliki je broj kriterija optimizacije te pri tome uključiti metrike koje mjere međusobno udaljenost i raznolikost rješenja [67]. Stoga su za evaluaciju odabrane sljedeće tri mjere:

- hipervolumen - HV (engl. *hypervolume*),
- inverzna međugeneracijska udaljenost - IGD (engl. *inverted generational distance*),
- korelacija sa referentnom frontom.

Hipervolumen je mjera koja računa volumen hiperprostora zauzetog Pareto frontom koju čine Pareto optimalna rješenja dobivena metodom koja se razmatra. Hiperprostor je omeđen Pareto frontom s jedne strane i točkom *nadir* s druge strane. Nadir se dobiva tako da se iz skupa svih rješenja (objedinjena rješenja svih metoda koje se uspoređuju) odabere najgore te mu se pridoda neki pomak *delta* kako bi i ekstremi tj. krajnje točke na fronti imale neki doprinos u računanju hipervolumena. Pri usporedbi dviju metoda boljom se smatra ona koja ima veći hipervolumen. Primjer računanja hipervolumena je dan na slici 4.17a. Ova metrika se često koristi jer je vrlo intuitivna za razumijevanje, neovisna o skaliranju i jednostavno prikazuje razlike u performansama dviju fronti [68].

Inverzna međugeneracijska udaljenost je srednja udaljenost svake točke (rješenja), iz *referentne fronte* do najbliže točke u Pareto fronti skupa rješenja za koji se računa IGD, kao što je prikazano na slici 4.17b. Točke u referentnoj fronti su globalno Pareto optimalna rješenja. Budući da je problem DSE-a za heterogene višeprocesorske sustave NP-težak, globalna Pareto fronta nije poznata pa se referentna fronta dobiva tako što se za ispitni slučaj objedine sva rješenja dobivena svim postupcima koji se međusobno uspoređuju te se iz tog skupa izdvoji Pareto fronta koja onda postaje referentna fronta. Pri usporedbi dvije metode boljom se smatra ona koja ima manji IGD što znači da su rješenja iz Pareto fronte dobivene tom metodom vrlo blizu svake točke iz referentne fronte [69].



(a) Hipervolumen (HV)

(b) Inverzna generacijska udaljenost (IGD)

Slika 4.17: Prikaz načina izračuna mjera za evaluaciju optimizacijskih metoda

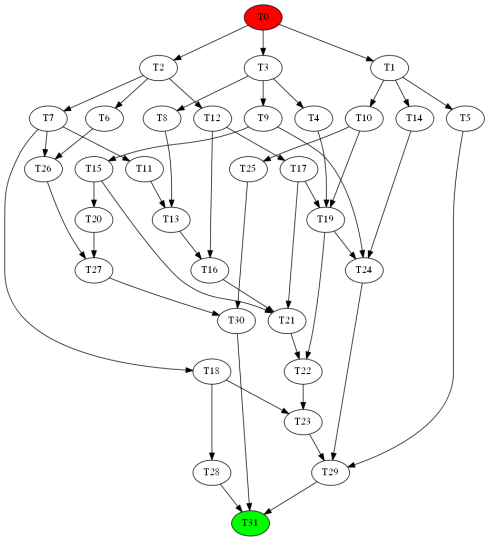
Kao zadnja mjera, odabrana je korelacija Pareto fronte skupa rješenja kojeg se dobije pojedinom metodom sa *referentnom frontom*. Korelacija se mjeri tako da se izračuna koliko udio u rješenjima iz referentne fronte čine rješenja iz Pareto fronte dobivene metodom koja se evaluira.

4.4.1 Umjetni modeli

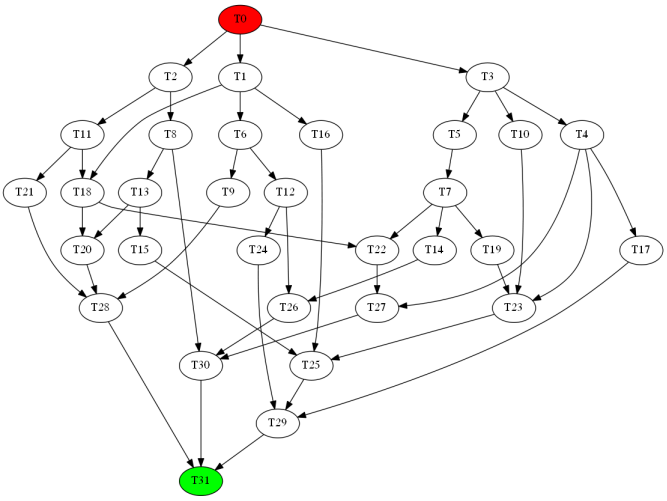
PSPLib je biblioteka skupova problema namijenjenih za ispitivanje postupaka raspoređivanja nad ograničenim resursima (engl. *resource constrained scheduling*). Iz te biblioteke odabran je skup *j30.sm* u kojem se nalazi 480 nasumično generiranih modela, svaki sa po 30 zadaća koje se izvode na 4 različita izvršna čvora. Zadaće su međusobno povezane kao čvorovi u usmjerenom grafu (primjeri na slici 4.18). Maksimalni broj zadaća koje se izvode u paraleli varira od 7 do 10, a broj razina čvorova u grafu je u rasponu od 9 do 11. Svaka zadaća za svoje izvođenje zahtijeva određenu količinu resursa: od najmanje jednog do najviše četiri izvršna čvora, pri čemu se mogu identificirati tri podskupine:

1. podskupina 1 - svaka zadaća koristi samo jedan procesor
2. podskupina 2 - svaka zadaća koristi od dva do najviše tri procesora
3. podskupina 3 - gotovo svaka zadaća koristi sva četiri procesora.

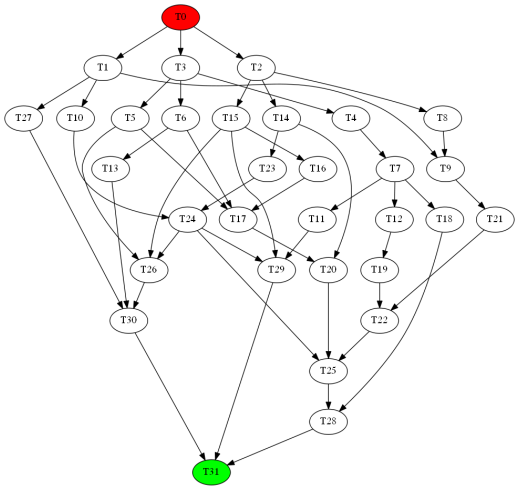
U nastavku će se model kakav je dan u skupu j30.sm nazivati - *PSPLib model*.



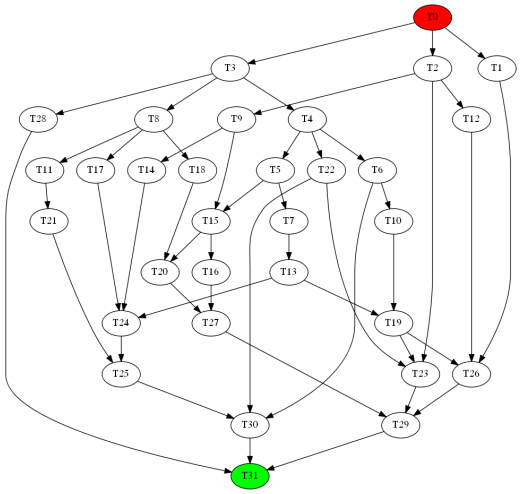
(a) j301_1



(b) j305_4



(c) j3010_1



(d) j3014_4

Slika 4.18: Grafovi povezanosti zadaća u PSPLib modelima

Na temelju PSPLib modela izrađeni su modeli platforma i aplikacija kakve koristi predložena metoda pretrage prostora oblikovanja. Radi jasnoće i razlikovanja izrađeni modeli se u nastavku nazivaju - *DSE model platforme* i *DSE model aplikacije*.

Pri izradi DSE modela na temelju PSPLib modela, zadaće postaju procedure, a izvršni čvorovi postaju procesori. DSE model platforme se iz PSPLib modela dobiva tako što se skup izvršnih čvorova koje koriste zadaće u PSPLib modelu pretvara u skup procesora na kojima izvode procedure u DSE modelu platforme. Budući da svaki PSPLib model sadrži točno 4 vrste izvršnih čvorova, iz svakog PSPLib modela se može izgraditi DSE model platforme s četiri različite vrste procesora. Vrijeme trajanja izvođenja pojedine procedure na pojedinom procesoru se

dobiva iz PSPLib modela tako da se količina resursa koju jedna zadaća zauzme na konkretnom procesoru pomnoži s 10^{-6} što onda približno odgovara redu veličine trajanja izvođenja srednje zahtjevne procedure u ugradbenim sustavima. Budući da u DSE modelu platforme za svaki procesor mora biti definirano trajanje izvođenja svih procedura u aplikaciji, za one procedure koje se ne mogu izvesti na tom procesoru se definira beskonačno trajanje: ∞.

U PSPLib modelu ne postoje memorijski niti komunikacijski elementi, ali se smatra da su svi procesori međusobno direktno povezani. Iz toga proizlazi da u DSE modelu platforme svi procesori trebaju biti međusobno povezani, a memorijski elementi se mogu modelirati na temelju karakteristika tipičnih memorijskih elemenata kakvi se mogu pronaći na ugradbenim razvojnim platformama poput Zynq [8] i Parallella [9] platformi.

DSE model aplikacije se iz PSPLib modela dobiva tako što zadaće iz PSPLib modela postaju procedure u DSE modelu. Usmjereni graf povezanosti zadaća iz PSPLib modela se direktno preslikava u DAG procedura u DSE modelu aplikacije. Međutim, zadaće u PSPLib modelu međusobno ne razmjenjuju podatke te stoga nije moguće direktno iz toga modela izvesti komunikaciju potrebnu za DSE model. Budući da je komunikacija važan element u heterogenim višeprocesorskim sustavima, ona se uvodi na sljedeći način. Komunikacija se odvija samo između onih procedura koje su međusobno povezane u DAG-u i to u smjeru veze. Količina podataka koja se razmjenjuje između dvije procedure određena je na temelju omjera trajanja komunikacije i izračuna - *CCR* (engl. *communication-to-computation ratio*). CCR je standardni parametar koji se koristi pri sintezi umjetnih ispitnih slučajeva u kojima je aplikacija zadana kao DAG [59], a težine čvorova i bridova, koji predstavljaju trajanje izračuna odnosno komunikacije, generiraju se nasumično poštujući jednoliku distribuciju i unaprijed zadani CCR. Za svaki par procedura količina podataka koja se razmjenjuje računa se na temelju unaprijed zadanog CCR-a i podataka iz DSE modela platforme kao što je prosječno trajanje izvođenja procedura te prosječna propusnost memorije prema formuli:

$$kol\_podataka = \frac{prosjecno\_trajanje\_izvodenja\_procedura}{prosjecna\_propusnost\_memorije} * CCR \tag{4.8}$$

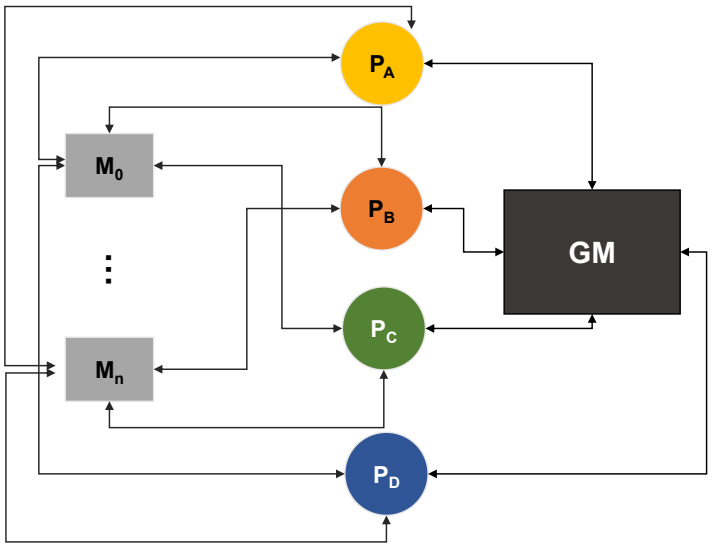
Dobivena vrijednost se zatim množi s nasumičnim brojem iz intervala [0.8, 1.2] kako bi se izbjeglo da sve procedure razmjenjuju istu količinu podataka.

Ispitni slučajevi

Za izradu ispitnih slučajeva odabrani su sljedeći konkretni modeli iz svake podskupine s obzirom na broj izvršnih čvorova koji koriste: *j301_1* iz podskupine 1, *j305_4* iz podskupine 2 te *j3010_1* i *j3014_4* iz podskupine 3. Model j3014_4 je izabran jer u njemu apsolutno svaka zadaća koristi sva četiri procesora. Grafovi povezanosti zadaća u modelima su prikazani na slici 4.18.

Na temelju svakog pojedinog PSPLib modela napravljeno je 7 DSE modela aplikacije u kojima je zadržana ista struktura međuovisnosti (povezanosti) procedura u aplikaciji, a mijenjana je količina komunikacije, tj. faktor CCR. Budući da komunikacija u heterogenim sustavima ima velik utjecaj na performanse, odabran je širok raspon vrijednosti faktora CCR: 0.01, 0.1, 0.5, 1, 5, 10 i 20, kako bi se pokrili slučajevi male, velike i podjednake količine komunikacije u odnosu na izračun.

DSE modeli platforme su izgrađeni tako da su na temelju svakog od prethodnog navedenih 4 PSPLib modela napravljena dva DSE modela platforme: model sa 4 instance od svake vrste procesora - *Platform16a* i model s tri instance od svake vrste procesora - *Platform12a*. Uz to u svakom DSE modelu platforme dodana je jedna velika i spora memorija koja po performansama približno odgovara DDR memoriji na tipičnoj Xilinx Zynq platformi te određeni broj manjih i brzih memorija koje po performansama približno odgovaraju BRAM memoriji na Zynq platformama: u modelu *Platform16a* postoje četiri takve memorije, a u modelu *Platform12a* tri. Svi procesori su povezani sa svim memorijama. Zbog nepreglednosti prikaza veza svih 12 odnosno 16 procesora sa svim memorijama, na slici 4.19 je prikazan graf smanjenog modela platforme koji ima 4 procesora, po jedna instanca od svake vrste. Memorije označene na grafu sa M_0 - M_n predstavljaju male i brze memorije, a memorija označena sa *GM* veliku i sporu memoriju.



Slika 4.19: Primjer DSE modela platforme s po jednom instancom od svake vrste procesora

Sveukupno je za ispitivanje pripremljeno 56 konfiguracija sustava na temelju umjetnih modela. Konfiguracije umjetnih modela su stvorene tako što je su skupine od po 7 modela aplikacije s istom strukturom, a različitim faktorom CCR kombinirane sa pripadajućim modelima platformi *Platform16a* i *Platform12a*. Nad opisanim konfiguracijama pokrenuti su algoritmi pretrage prostora oblikovanja SDSE i 2SDSE i to po 300 pokretanja svakog algoritma za svaku konfiguraciju.

Pri obradi rezultata, rješenja pronađena u svih 300 pokretanja jednog algoritma nad jednom konfiguracijom objedinjena su u jedinstven skup iz kojeg je zatim izdvojena Pareto fronta s najboljim rješenjima za taj skup - *referentna fronta*. Naravno, to ne znači da su to i globalno Pareto optimalna rješenja zato što kod NP-teških problema Pareto optimalna rješenja vrlo često nije ni moguće točno odrediti.

U tablicama 4.3, 4.4 i 4.5 prikazani su svi rezultati za umjetno generirane modele i to za mjere hipervolumen, IGD i korelaciju s referentnom frontom. Pri izračunu hipervolumena i IGD-a bilo je potrebno normalizirati dobivena rješenja budući da je vremensko trajanje reda veličine 10^{-4} a broj komponenti reda veličine 10^{1}. Na slikama 4.20, 4.21 i 4.22 prikazana je usporedba istovremene i slijedne inačice metode pretrage prostora oblikovanja za svih 56 konfiguracija i sve tri mjere.

Tablica 4.3: Rezultati za umjetno generirane modele - hipervolumen

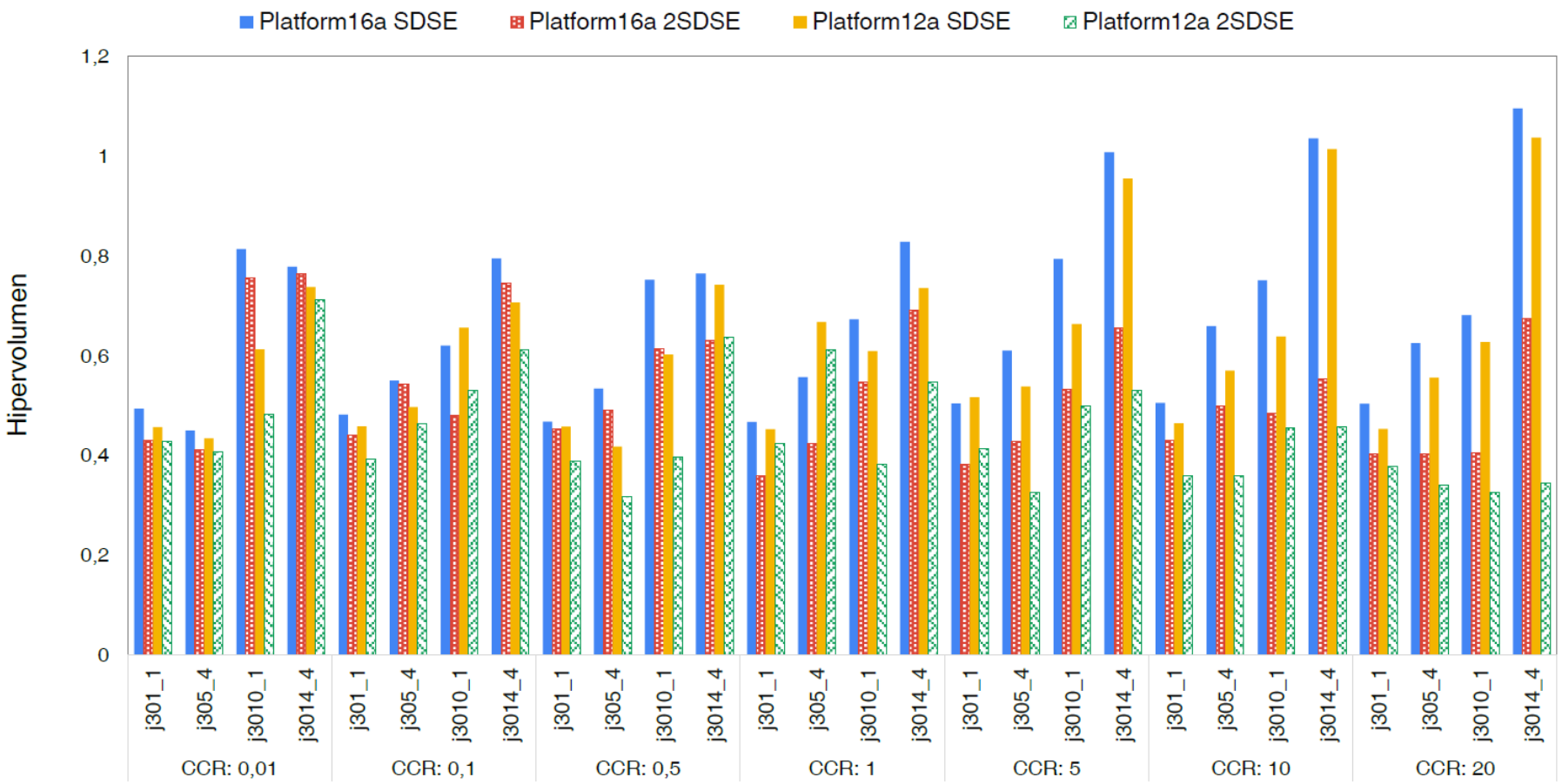
		PLATFORM16A				PLATFORM12A			
		j301_1	j305_4	j3010_1	j3014_4	j301_1	j305_4	j3010_1	j3014_4
CCR: 0.01	SDSE	0.49	0.45	0.81	0.78	0.46	0.43	0.61	0.74
	2SDSE	0.43	0.41	0.75	0.76	0.43	0.41	0.48	0.71
CCR: 0.1	SDSE	0.48	0.55	0.62	0.79	0.46	0.50	0.66	0.71
	2SDSE	0.44	0.54	0.48	0.75	0.39	0.46	0.53	0.61
CCR: 0.5	SDSE	0.47	0.53	0.75	0.76	0.46	0.42	0.60	0.74
	2SDSE	0.45	0.49	0.61	0.63	0.39	0.32	0.40	0.64
CCR: 1	SDSE	0.47	0.56	0.67	0.83	0.45	0.67	0.61	0.74
	2SDSE	0.36	0.42	0.55	0.69	0.42	0.61	0.38	0.55
CCR: 5	SDSE	0.50	0.61	0.79	1.01	0.52	0.54	0.66	0.95
	2SDSE	0.38	0.43	0.53	0.65	0.41	0.33	0.50	0.53
CCR: 10	SDSE	0.50	0.66	0.75	1.03	0.46	0.57	0.64	1.01
	2SDSE	0.43	0.50	0.49	0.55	0.36	0.36	0.45	0.46
CCR: 20	SDSE	0.50	0.62	0.68	1.10	0.45	0.56	0.63	1.04
	2SDSE	0.40	0.40	0.40	0.67	0.38	0.34	0.32	0.34

Tablica 4.4: Rezultati za umjetno generirane modele - IGD

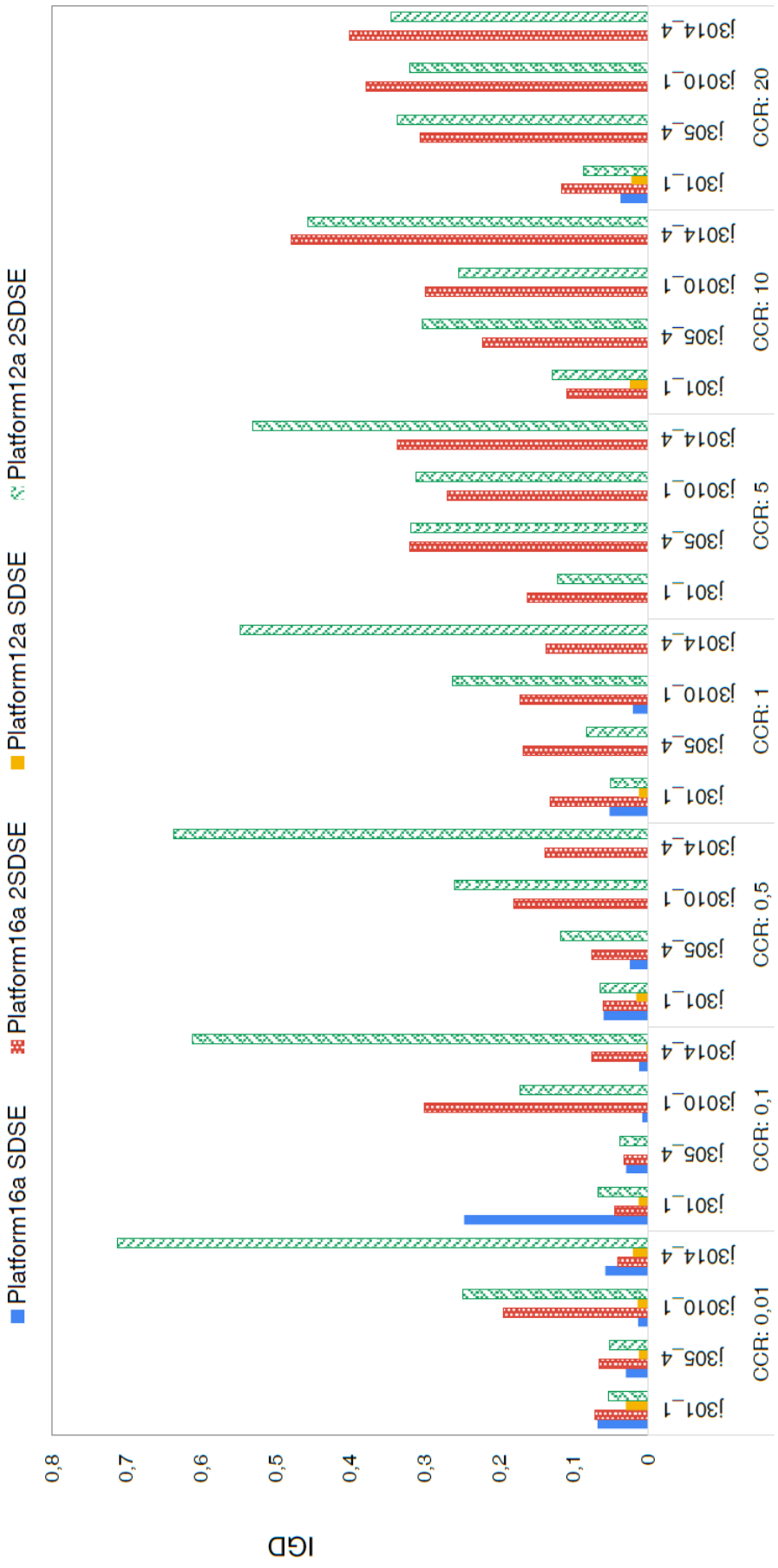
		PLATFORM16A				PLATFORM12A			
		j301_1	j305_4	j3010_1	j3014_4	j301_1	j305_4	j3010_1	j3014_4
CCR: 0.01	SDSE	6.70E-02	2.91E-02	1.27E-02	5.66E-02	2.89E-02	1.16E-02	1.34E-02	1.95E-02
	2SDSE	7.09E-02	6.46E-02	1.94E-01	3.93E-02	5.22E-02	5.10E-02	2.48E-01	7.12E-01
CCR: 0.1	SDSE	2.46E-01	2.87E-02	7.19E-03	1.12E-02	1.20E-02	0.00E+00	0.00E+00	1.41E-03
	2SDSE	4.45E-02	3.11E-02	3.00E-01	7.45E-02	6.69E-02	3.66E-02	1.71E-01	6.12E-01
CCR: 0.5	SDSE	5.91E-02	2.39E-02	0.00E+00	0.00E+00	1.49E-02	0.00E+00	0.00E+00	0.00E+00
	2SDSE	6.02E-02	7.52E-02	1.80E-01	1.38E-01	6.34E-02	1.17E-01	2.59E-01	6.37E-01
CCR: 1	SDSE	5.12E-02	0.00E+00	1.94E-02	0.00E+00	1.17E-02	0.00E+00	0.00E+00	0.00E+00
	2SDSE	1.31E-01	1.67E-01	1.71E-01	1.36E-01	4.96E-02	8.17E-02	2.62E-01	5.46E-01
CCR: 5	SDSE	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00
	2SDSE	1.62E-01	3.20E-01	2.69E-01	3.37E-01	1.21E-01	3.18E-01	3.11E-01	5.30E-01
CCR: 10	SDSE	0.00E+00	0.00E+00	0.00E+00	0.00E+00	2.40E-02	0.00E+00	0.00E+00	0.00E+00
	2SDSE	1.08E-01	2.21E-01	2.98E-01	4.79E-01	1.27E-01	3.03E-01	2.53E-01	4.57E-01
CCR: 20	SDSE	3.64E-02	0.00E+00	0.00E+00	0.00E+00	2.17E-02	0.00E+00	0.00E+00	0.00E+00
	2SDSE	1.15E-01	3.06E-01	3.78E-01	4.00E-01	8.67E-02	3.36E-01	3.19E-01	3.44E-01

Tablica 4.5: Rezultati za umjetno generirane modele - korelacija s referentnom Pareto frontom

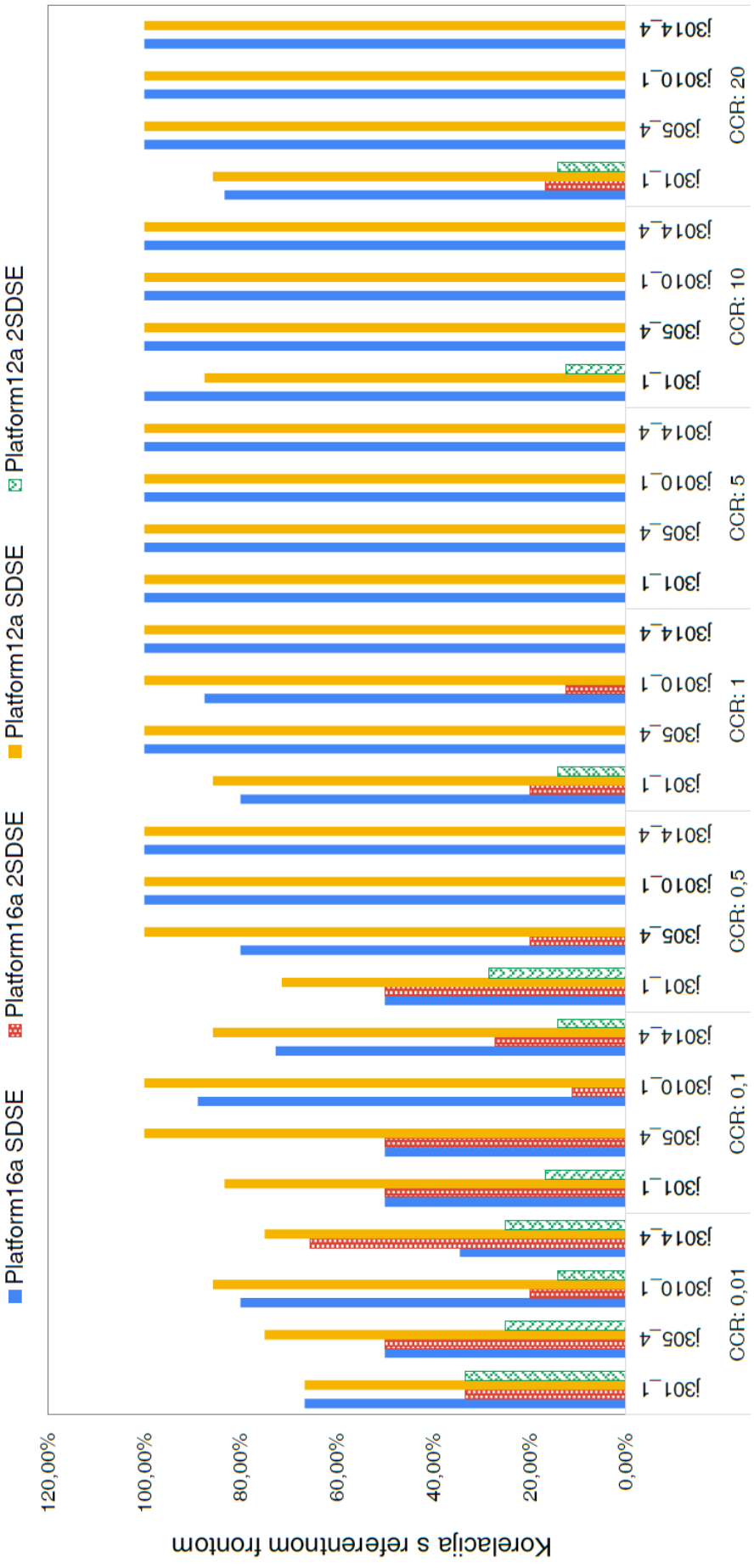
		PLATFORM16A				PLATFORM12A			
		j301_1	j305_4	j3010_1	j3014_4	j301_1	j305_4	j3010_1	j3014_4
CCR: 0.01	SDSE	0.67	0.50	0.80	0.35	0.67	0.75	0.86	0.75
	2SDSE	0.33	0.50	0.20	0.66	0.33	0.25	0.14	0.25
CCR: 0.1	SDSE	0.50	0.50	0.89	0.73	0.83	1.00	1.00	0.86
	2SDSE	0.50	0.50	0.11	0.27	0.17	0.00	0.00	0.14
CCR: 0.5	SDSE	0.50	0.80	1.00	1.00	0.71	1.00	1.00	1.00
	2SDSE	0.50	0.20	0.00	0.00	0.29	0.00	0.00	0.00
CCR: 1	SDSE	0.80	1.00	0.88	1.00	0.86	1.00	1.00	1.00
	2SDSE	0.20	0.00	0.13	0.00	0.14	0.00	0.00	0.00
CCR: 5	SDSE	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	2SDSE	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
CCR: 10	SDSE	1.00	1.00	1.00	1.00	0.88	1.00	1.00	1.00
	2SDSE	0.00	0.00	0.00	0.00	0.13	0.00	0.00	0.00
CCR: 20	SDSE	0.83	1.00	1.00	1.00	0.86	1.00	1.00	1.00
	2SDSE	0.17	0.00	0.00	0.00	0.14	0.00	0.00	0.00



Slika 4.20: Usporedba rezultata za umjetno generirane modele - hipervolumen



Slika 4.21: Usporedba rezultata za umjetno generirane modele - IGD



Slika 4.22: Usporedba rezultata za umjetno generirane modele - korelacija s referentnom frontom

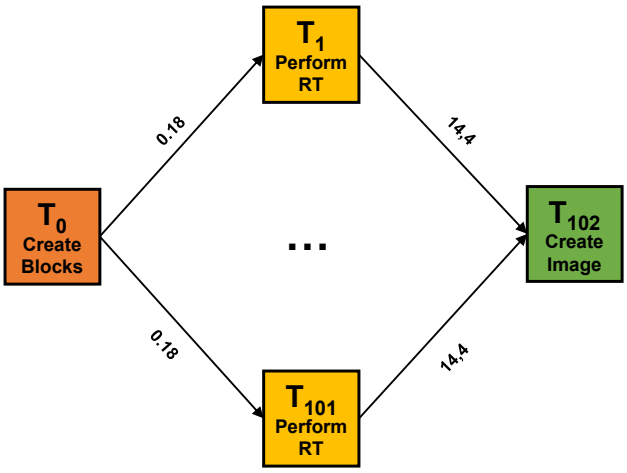
Analiza rezultata pokazuje da je algoritam SDSE bolji od algoritma 2SDSE u većini slučajeva i to za sve tri metrike. Gledajući isključivo hipervolumen, algoritam SDSE daje bolja rješenja od algoritma 2SDSE za apsolutno sve konfiguracije. Za konfiguracije j301_1 i j305_4 razlika između ova dva algoritma je općenito nešto manja nego što je za konfiguracije j3010_1 i j3014_4. Najmanja razlika je za slučaj j305_4 s faktorom CCR = 0.1 gdje je hipervolumen rješenja dobivenih algoritmom SDSE za svega 1,2% veći od onog za rješenja dobivenih algoritmom 2SDSE. Razlog je to što se u slučaju konfiguracija j301_1 i j305_4 procedure ne mogu izvesti na svim procesorima, čime se znatno smanjuje skup mogućih rješenja. Nadalje, vidljivo je da što je veći faktor CCR, to i algoritam 2SDSE daje znatno lošija rješenja od algoritma SDSE. Najveća razlika je za slučaj j3014_4 i CCR = 20 gdje je hipervolumen za SDSE 202% bolji od hipervolumena za 2SDSE. To se može objasniti činjenicom da što je veći faktor CCR, to je utjecaj komunikacije na trajanje izvođenja znatno veći pa time i odabir rasporeda koji je nepovoljniji za komunikaciju znatnije utječe na kvalitetu rješenja.

Prema mjeri IGD, algoritam SDSE daje bolja rješenja u 91% slučajeva, ali postoje iznimke za konfiguracije j301_1 s malim faktorom CCR gdje algoritam 2SDSE daje jednako dobar (5% slučajeva) ili čak nešto bolji rezultat u usporedbi s algoritmom SDSE (4% slučajeva). Tako u slučaju j3014_4 za CCR = 0.01 SDSE ima 44% veću vrijednost IGD-a od 2SDSE-a, a u slučaju j301_1 za CCR = 0.1 SDSE ima 4.5 puta veću vrijednost IGD-a od 2SDSE-a. Slično pokazuje i mjera korelacije s referentnom frontom što se ponovno može objasniti smanjenjem skupa mogućih rješenja i manjim utjecajem komunikacijski nepovoljnijeg rasporeda kod konfiguracija s vrlo malom količinom komunikacije.

4.4.2 Stvarne aplikacije i platforme

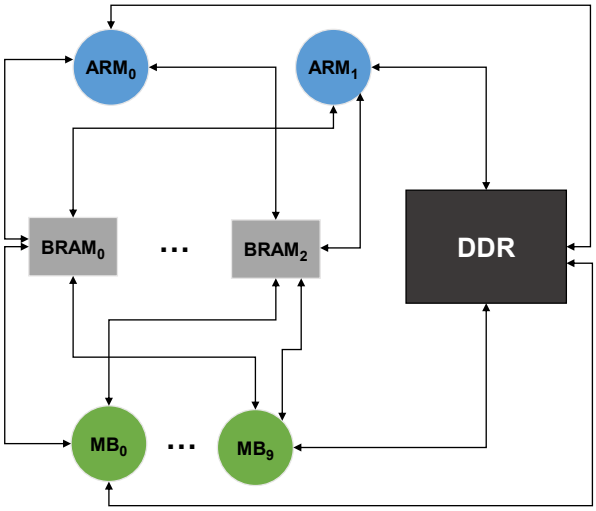
Za evaluaciju algoritama SDSE i 2SDSE korištene su i stvarne aplikacije: kompresija JPEG i praćenje zrake - RT (engl. *Ray Tracing*) na platformama Xilinx ZC706 i Adapteva Parallela. Kompresija JPEG je modelirana za sliku Lenna veličine 40 blokova od 64x64 bita. Model aplikacije je isti kao što je prikazano na slici 4.3 samo što u ovom slučaju postoji 40 blokova pa je broj čvorova koji predstavljaju procedure *Shift*, *DCT* i *ZigZag* 20 puta veći. Aplikacija RT je modelirano za sliku veličine 800x600 piksela na kojoj se iscrtavaju tri kugle polumjera 100 piksela. Slika je podijeljena u dijelove veličine 80x60 piksela koji se iscrtavaju u paraleli. RT aplikacija se sastoji od procedure *CreateBlocks* koja stvara blokove, 100 paralelnih poziva procedure za iscrtavanje jednog bloka *PerformRT* i konačno procedure *CreateImage* koja spaja

blokove u sliku. Model aplikacije RT je prikazan pomoću DAG-a na slici 4.23.

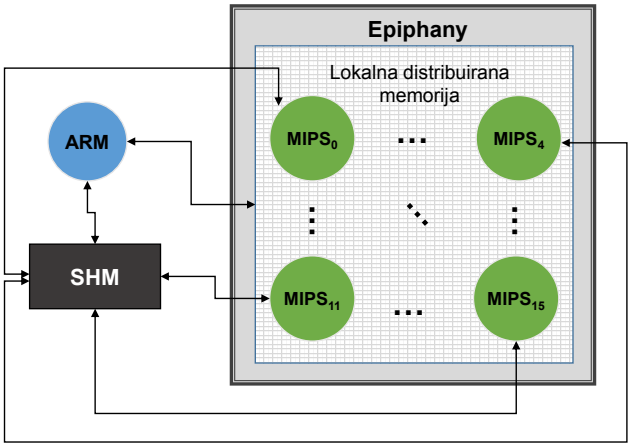


Slika 4.23: DAG aplikacije RT

Model platforme Xilinx ZC706 se sastoji od dva ARM procesora, 10 Microblaze procesora, jedne velike i spore DDR memorije te 3 male i brze BRAM memorije. Svi procesori su spojeni na sve memorije kao što je prikazano na slici 4.24. Model platforme Adapteva Parallela sadrži jedan ARM procesor, 16 MIPS jezgri na Epiphany čipu te dvije memorije. Jedna memorija je dijeljena između procesora i Epiphany čipa i naziva se SHM. Druga memorija je lokalna memorija koju dijele sve MIPS jezgre na Epiphany čipu. Procesor ARM može pristupati objema memorijama s time da je pristup memoriji SHM za dva reda veličine brži. MIPS procesori na Epiphany čipu također mogu pristupati objema memorijama, međutim lokalna memorija na Epiphany čipu je ujedno i memorija u kojoj MIPS jezgre drže instrukcije i podatke nad kojima trenutno rade i sav sadržaj je jednako dostupan svim jezgrama što znači da si jezgre ne moraju slati podatke pa je trajanje razmjene podataka između dviju jezgri jednako "*0*". Graf modela Parallela platforme je prikazan na slici 4.25.



Slika 4.24: Graf modela platforme Xilinx ZC706



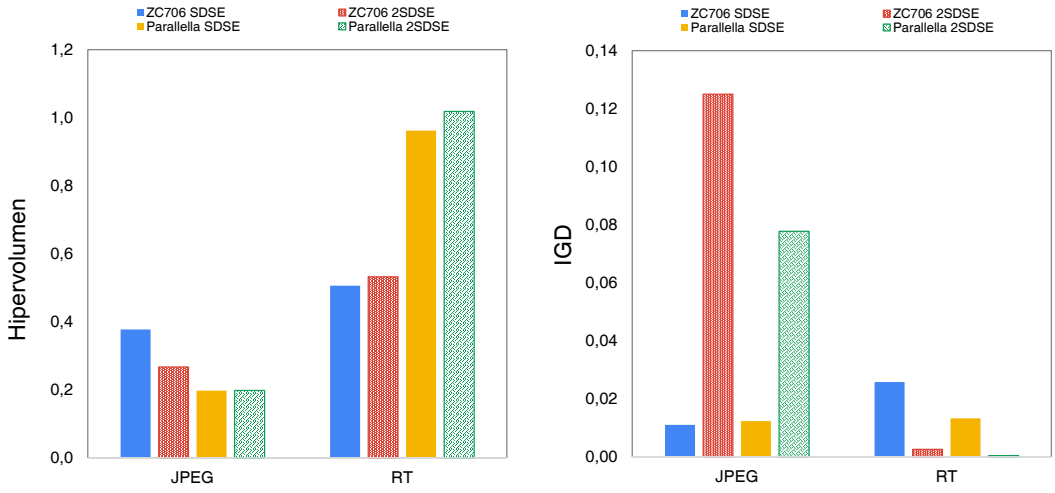
Slika 4.25: Graf modela platforme Adapteva Parallella

U slučaju aplikacije JPEG, sve procedure se mogu izvesti na svim procesorima, a faktor CCR iznosi približno 0.02, što je slično s konfiguracijom j3014_4 za CCR 0.01 samo s mnogo većim brojem procedura (363). U slučaju aplikacije RT, prva i zadnja procedura mogu izvesti samo na procesoru ARM na obje platforme, a faktor CCR iznosi približno 0.03, što je slično s j301_1 za CCR 0.01 samo s većim brojem procedura (102). Faktor CCR je za obje aplikacije određen približno, tako da se odredilo trajanje ukupne komunikacije i ukupnog izvođenja za svaku memoriju, odnosno procesor, iz čega je izračunato prosječno trajanje ukupne komunikacije i prosječno trajanje ukupnog izvođenja, te CCR kao njihov omjer.

Ispitni slučajevi

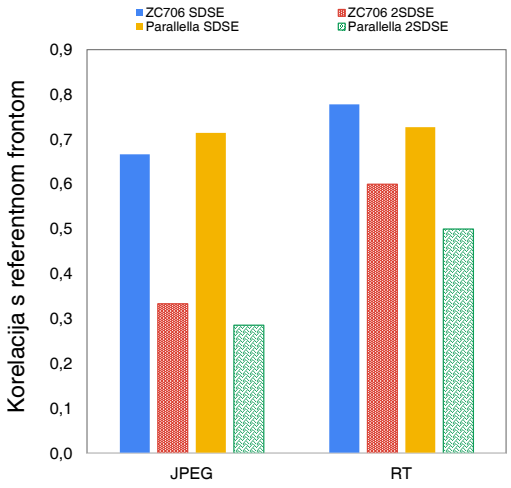
Ispitivanje je provedeno na 4 konfiguracije stvarnih modela: aplikacije JPEG i RT su kombinirane s modelima platforma ZC706 i Parallella. Nad opisanim konfiguracijama pokrenuti su algoritmi pretrage prostora oblikovanja SDSE i 2SDSE i to po 300 pokretanja svakog algoritma za svaku konfiguraciju.

Usporedba rezultata za aplikacije JPEG i RT na platformama ZC706 i Parallella prikazana je na slici 4.26. Pri izračunu hipervolumena i IGD-a ponovno je bilo potrebno normalizirati dobivena rješenja budući da je vremensko trajanje reda veličine 10^{-4} a broj komponenti reda veličine 10^{1}.



(a) Hipervolumen

(b) IGD



(c) Korelacija s referentnom frontom

Slika 4.26: Usporedba hipervolumena, IGD-a i korelacije s referentnom frontom za aplikacije JPEG i RT na platformama Xilinx ZC706 i Adapteva Parallella

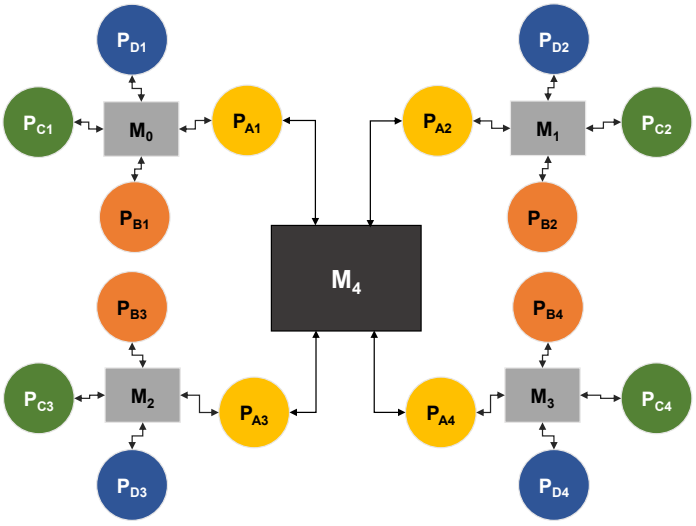
U slučaju aplikacije JPEG, za sve tri mjere i za sve konfiguracije algoritam SDSE je bolji od algoritma 2SDSE. U slučaju aplikacije RT, algoritam SDSE je prema mjeri korelacije bolji od algoritma 2SDSE, dok su prema mjeri hipervolumena oba algoritma podjednaka, a prema mjeri IGD algoritam SDSE je za red veličine lošiji od algoritma 2SDSE. No, treba također naglasiti da kad se pogleda konkretna rješenja dobivena ovim postupcima i pojedinačni kriteriji, trajanje izvođenja rješenja u Pareto fronti se između ova dva postupka ne razlikuje za više od ~10%, dok broj korištenih elementa se razlikuje za 1 do 2 elementa. Pri traženju objašnjenja zašto nije uočena značajnija razlika između ova dva algoritma u slučaju aplikacije RT, potrebno je istaknuti da je aplikacija RT vrlo specifičan problem. Naime, ova aplikacija se izvodi u samo tri slijedna stupnja od čega prvi i treći imaju samo po jednu proceduru. Upravo te dvije procedure proizvode i najviše komunikacije, a mogu se izvesti na samo jednom procesoru. To znači da za pridruživanje te dvije procedure na procesore postoji samo jedno rješenje do kojeg moraju doći oba algoritma. Jedina moguća varijacija je u pridruživanju procedura koje se izvode u paraleli u drugom stupnju, a tu je prisutna vrlo mala količina komunikacije.

Ukupno gledano za stvarne i umjetne modele, algoritam SDSE je bez sumnje znatno uspješniji od algoritma 2SDSE. U pojedinim slučajevima oba algoritma daju podjednake rezultate, no SDSE ni u jednom slučaju ne daje znatno lošije rezultate od algoritma 2SDSE. To znači da je istovremena pretraga prostora oblikovanja za procesore i memorije uspješnija metoda. Rezultati su objavljeni 2018. godine [70].

4.5 Problem rijetko povezane platforme

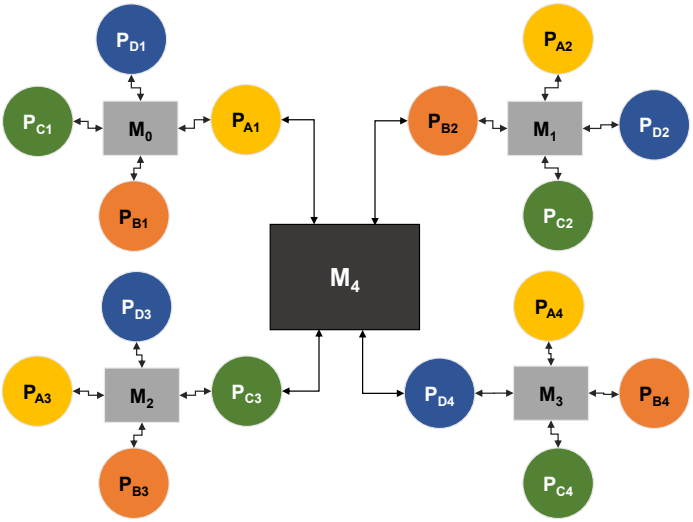
Stvarne višeprocesorske platforme nisu nužno uvijek potpuno povezane, tj. nemaju svi procesori pristup svim memorijama. U takvim slučajevima problem pretrage prostora oblikovanja postaje još složeniji budući da je sve veća mogućnost dodjele procedura na procesore između kojih ne postoji veza što rezultira nemogućim (neizvedivim) rješenjem. Primjer modela jedne takve platforme je prikazan kao graf na slici 4.27. Ovaj model izveden je iz modela *Platform16a* tako što su procesori grupirani u skupine po 4 te svaka skupina sadrži po jedan procesor od svake vrste (ista vrsta procesora je označena na slici istim početnim slovom i istom bojom) i jednu malu memoriju na koju su povezani svi procesori u skupini. Samo jedan procesor iz svake skupine je povezan i s velikom memorijom. U ovom slučaju broj nemogućih znatno premašuje broj mogućih rješenja i postavlja se pitanje da li algoritam SDSE uopće može pronaći moguća

rješenja.



Slika 4.27: Primjer rijetko povezane platforme

Učinkovitost algoritma SDSE u slučaju rijetko povezane platforme provjerena je na 5 posebno stvorenih umjetnih modela. Prvi model - *Platform16c1* je prikazan na slici 4.27. U tom modelu su samo procesori P_{A1}, P_{A2}, P_{A3} i P_{A4}, svi iste vrste, povezani s dvije memorije dok su svi ostali procesori povezani samo s jednom memorijom. Na isti način načinjeni su modeli *Platform16c2*, *Platform16c3* i *Platform16c4* tako što je svaki put promijenjena vrsta procesora koji su spojeni na dvije memorije. Poseban slučaj je model *Platform16c5*, prikazan na slici 4.28, gdje je iz svake skupine odabrana različita vrsta procesora koji je spojen s dvije memorije.



Slika 4.28: Model Platform16c5

Rezultati izvođenja, prikazani u tablicama 4.6, 4.7 i 4.8 pokazuju da algoritam SDSE nije uvijek u stanju pronaći izvedivo rješenje - prazna mjesta u tablicama znače da nije pronađeno nijedno izvedivo rješenje. Stoga je napravljeno nekoliko inačica algoritma kako bi se pokušalo postići bolje rezultate. Algoritam 2SDSE nije razmatran za slučaj rijetko povezanih platforman budući da kod ovih modela rijetko povezane platforme ne postoji više od jednog puta između dva procesora. Samim time je zapravo svejedno odvija li se pridruživanje komunikacijskih kanala na memorije istovremeno ili u zasebnoj fazi, a tehnički je SDSE jednostavniji za izvedbu jer nije potrebna obrada rezultata između dvije faze i ponovno pokretanje kao što je slučaj kod algoritma 2SDSE.

U inačici N-SDSE (*Non-Infinity SDSE*)[d] osnovni algoritam SDSE je modificiran na način da trajanje izvođenja nemogućih rješenja više ne iznosi beskonačno nego se za svaki par zadaća koje međusobno komuniciraju, a dodijeljene su na procesore između kojih ne postoji veza, pribraja tzv. "kazna" u iznosu od 10^{10}. Na taj način se uvodi gradacija loših rješenja tj. omogućava razlikovanje rješenja koja u sebi imaju manji broj nemogućih dodjela procedura na procesore i koja bi potencijalno kroz manji broj križanja i/ili mutacija mogla postati izvediva rješenja.

Inačica C-SDSE (engl. *Clustered SDSE*)[e] uvodi modifikaciju u proces dekodiranja gena u kromosomu, tj. određivanja procesora na kojima će se izvesti pojedina procedura. Platforma se najprije analizira na način da se detektiraju skupine procesora koji su povezani na istu memoriju - *grozdovi*. Nakon što su određeni grozdovi procesora na platformi, pridruživanje procedura procesorima (tj. dekodiranje gena u kromosomu) se odvija u dva koraka. Najprije se pronalaze grozdovi u kojima postoji barem jedan procesor na kojem se može izvesti zadaća i na temelju vrijednosti gena se odabire grozd, a zatim se, ponovno na temelju vrijednosti gena, odabire procesor iz grozda (u obzir dolaze samo procesori koji mogu izvesti zadanu proceduru). Kako bi se pokušalo izbjeći nemoguća rješenja u što većoj mjeri, ako su procedura i njen sljedbenik na različitim grozdovima, pokušava se pronaći procesor koji se nalazi u oba grozda. Ako to nije moguće, na temelju vrijednosti gena se odabire procesor iz grozda koji može izvesti proceduru. Pseudokôd postupka pridruživanja procedura procesorima je prikazan na slici 4.29.

Inačica CN-SDSE (engl. *Clustered Non-Infinity SDSE*)[f] funkcionira na isti način kao i C-SDSE, s time da trajanje izvođenja nemogućih rješenja više ne iznosi beskonačno nego se za svaki par zadaća koje međusobno komuniciraju, a dodijeljene su na procesore između kojih ne

[d] Izvorni kôd je dostupan na adresi `https://gitlab.com/Frid/dsexplorer/tree/master/N-SDSE`
[e] Izvorni kôd je dostupan na adresi `https://gitlab.com/Frid/dsexplorer/tree/master/C-SDSE`
[f] Izvorni kôd je dostupan na adresi `https://gitlab.com/Frid/dsexplorer/tree/master/CN-SDSE`

postoji veza pribraja tzv. "kazna" u iznosu od 1^{10}.

Algorithm 5 Pridruživanje procedura procesorima u algoritmu C-SDSE

```
1:  for procedura ∈ u aplikacija do
2:      podrzaniGrozdovi = pronađi sve grozdove u kojima postoji barem jedan procesor na kojem
        se može izvesti procedura
3:      grozd = odaberi grozd iz podrzaniGrozdovi na temelju vrijednosti gena
4:      pridruži procedura grozdu
5:  end for
6:  for procedura ∈ u aplikacija do
7:      for sljedbenik ∈ procedura− > sljedbenici do
8:          if procedura i sljedbenik na istom grozdu then
9:              podrzaniProc = pronađi sve procesore iz grozda na koji je pridružena procedura i na
                kojima se ona može izvesti
10:             proc = odaberi procesor iz podrzaniProc na temelju vrijednosti gena
11:         else
12:             kandidati = procesori koji se nalaze i u grozdu kojem je pridružena procedura i grozdu
                kojem je pridružen sljedbenik
13:             podrzaniProc = pronađi sve procesore iz kandidati na kojima se može izvesti
                procedura
14:             if podrzaniProc nije prazan skup then
15:                 proc = dekodiraj procesor iz podrzaniProc na temelju vrijednosti gena
16:             else
17:                 podrzaniProc pronađi sve procesore iz grozd na kojima se može izvesti procedura
18:                 proc = dekodiraj procesor iz podrzaniProc na temelju vrijednosti gena
19:             end if
20:         end if
21:     end for
22:     pridruži procedura prcesoru proc
23: end for
```

Slika 4.29: Pseudokôd postupka pridruživanja procedura procesorima u algoritmu C-SDSE

Posljednja inačica, MCN-SDSE (engl. *Main Cluster Non-Infinity SDSE*)[g] je modifikacija inačice CN-SDSE posebno prilagođena strukturi platforme kakva je prikazan u modelima Platform16c1 - Platform16c5. U spomenutim modelima uočljivo je postojanje *glavnog grozda*, tj. skupine procesora koja je spojena sa svim ostalim grozdovima. U ovoj inačici se pri odabiru procesora daje prednost procesorima koji mogu izvesti određenu proceduru, a ujedno su i dio glavnog grozda. Kako bi se izbjeglo da sve zadaće budu raspoređene isključivo na procesore u glavnom grozdu, za procedure kojima su svi prethodnici i sljdebenici u istom grozdu radi se preraspoređivanje na ostale procesore u istom grozdu koji mogu izvesti zadanu proceduru. Ovakav pristup bi trebao povećati vjerojatnost pronalaženja mogućeg rješenja za sve platforme, a osigurati 100% vjerojatnost pronalaženja rješenja u slučaju modela Platform16c5 gdje glavni

[g]Izvorni kôd je dostupan na adresi `https://gitlab.com/Frid/dsexplorer/tree/master/MCN-SDSE`

grozd sadrži sve vrste procesora što znači da se svaka procedura može izvesti na barem jednom procesoru u glavnom grozdu. Pseudokôd postupka pridruživanja procedura procesorima je prikazan na slici 4.30.

Algorithm 6 Pridruživanje procedura procesorima u algoritmu MCN-SDSE

```
1:  glavniGrozd = detektiraj glavni grozd
2:  for procedura ∈ u aplikacija do
3:      podrzaniGrozdovi = pronađi sve grozdove u kojima postoji barem jedan procesor na kojem
        se može izvesti procedura i koji imaju procesor u glavniGrozd
4:      if podrzaniGrozdovi prazan skup then
5:          podrzaniGrozdovi = pronađi sve grozdove u kojima postoji barem jedan procesor na ko-
            jem se može izvesti procedura
6:      end if
7:      grozd = odaberi grozd iz podrzaniGrozdovi na temelju vrijednosti gena
8:      pridruži proceduru grozdu
9:      podrzaniProc = pronađi sve procesore iz grozda kojem je pridružena procedura i na kojima
        se ona može izvesti
10:     for p ∈ podrzaniProc do
11:         if p je u glavniGrozd then
12:             dodaj p u kandidati
13:         end if
14:     end for
15:     if kandidati nije prazan skup then
16:         proc = dekodiraj procesor iz kandidati na temelju vrijednosti gena
17:     else
18:         proc = dekodiraj procesor iz podrzaniProc na temelju vrijednosti gena
19:     end if
20:     dodijeli proc proceduri
21: end for
22: for procedura ∈ u aplikacija do
23:     provjeri da li su svi prethodnici i sljedbenici od procedura u istom grozdu
24:     if jesu then
25:         podrzaniProc = pronađi sve procesore iz grozda kojem je pridružena procedura i na ko-
            jima se ona može izvesti
26:         proc = dekodiraj procesor iz podrzaniProc na temelju vrijednosti gena
27:     end if
28:     pridruži proceduru procesoru proc
29: end for
```

Slika 4.30: Pseudokôd postupka pridruživanja procedura procesorima u algoritmu MCN-SDSE

Usporedba svih pet inačica algoritma SDSE je prikazana u tablicama 4.6, 4.7 i 4.8.

Tablica 4.6: Usporedba svih pet inačica algoritma SDSE prema hipervolumenu

		Platform16c1					Platform16c2					Platform16c3					Platform16c4					Platform16c5				
		SDSE	C-SDSE	CN-SDSE	N-SDSE	MCN-SDSE	SDSE	C-SDSE	CN-SDSE	N-SDSE	MCN-SDSE	SDSE	C-SDSE	CN-SDSE	N-SDSE	MCN-SDSE	SDSE	C-SDSE	CN-SDSE	N-SDSE	MCN-SDSE	SDSE	C-SDSE	CN-SDSE	N-SDSE	MCN-SDSE
CCR: 0.01	j301_1	1.24E-02	1.00E-02				8.16E-02	6.65E-02				1.00E-02	1.00E-02				1.24E-02	1.00E-02				4.87E-02	3.95E-02			4.78E-02
	j305_4											1.00E-02											4.05E-01	3.91E-01	3.47E-01	4.07E-01
	j3010_1	4.26E-02					4.26E-02					5.56E-01	5.91E-01	5.92E-01		6.22E-01	5.89E-01				6.56E-01	5.64E-01	5.94E-01	5.96E-01	5.32E-01	5.96E-01
	j3014_4	6.99E-01	6.06E-01	5.99E-01	5.68E-01	6.29E-01	7.93E-01	7.06E-01			7.61E-01	6.63E-01	3.12E-01	3.67E-01		5.97E-01	6.69E-01	2.38E-01	3.36E-01		5.76E-01	6.48E-01	6.95E-01	6.94E-01	5.43E-01	7.07E-01
CCR: 0.1	j301_1	1.24E-02	1.00E-02				7.28E-02					1.00E-02					1.24E-02	1.00E-02				5.63E-02	4.72E-02			4.80E-02
	j305_4																					2.15E-01	2.03E-01	1.81E-01		1.89E-01
	j3010_1	1.59E-01					1.59E-01					2.23E-01		6.36E-01	6.47E-01	6.93E-01					5.89E-01	6.39E-01	6.26E-01	6.26E-01	5.16E-01	6.27E-01
	j3014_4	6.98E-01	6.24E-01	6.19E-01	6.99E-01	6.45E-01	7.61E-01	6.46E-01	3.48E-01		6.88E-01	6.65E-01	5.40E-01	3.14E-01		6.22E-01	7.25E-01	6.54E-01	4.26E-01		6.66E-01	6.39E-01	6.52E-01	6.53E-01	5.47E-01	6.80E-01
CCR: 0.5	j301_1	1.00E-02					1.31E-01	1.05E-01				1.24E-02	1.00E-02				1.24E-02	1.00E-02				9.61E-02	8.80E-02			4.80E-02
	j305_4	1.00E-02					3.95E-02					4.21E-02					1.91E-02					5.01E-01	4.02E-01	4.10E-01		4.00E-01
	j3010_1	3.29E-01				1.87E-01	1.42E-01				3.67E-02	5.48E-01	5.56E-01	5.59E-01	5.44E-01	6.17E-01	6.30E-01				6.28E-01	6.29E-01	5.39E-01	5.37E-01	6.14E-01	5.36E-01
	j3014_4	7.07E-01	7.79E-01	7.83E-01		8.51E-01	6.88E-01	8.25E-01	3.87E-01		8.54E-01	6.86E-01	7.94E-01	3.76E-01		8.31E-01	6.25E-01	5.71E-01	3.62E-01		7.91E-01	6.64E-01	7.10E-01	7.10E-01	5.94E-01	7.37E-01
CCR: 1	j301_1	1.26E-02	1.00E-02				7.83E-02	6.26E-02				1.24E-02	1.00E-02				1.26E-02	1.00E-02				3.20E-02				1.00E-02
	j305_4	1.00E-01				4.26E-02																4.45E-01	3.49E-01	3.49E-01		3.49E-01
	j3010_1	3.61E-01	7.63E-02			2.35E-01		1.31E-02				7.15E-01	6.12E-01	6.14E-01		7.19E-01	3.57E-01				3.88E-01	5.11E-01	5.34E-01	5.31E-01	5.08E-01	6.07E-01
	j3014_4	7.98E-01	9.56E-01	9.60E-01		9.79E-01	6.60E-01	8.42E-01	3.29E-01		8.72E-01	7.25E-01	7.03E-01	4.46E-01	7.12E-01	9.16E-01	6.70E-01	1.86E-01	4.17E-01		9.14E-01	9.39E-01	9.13E-01	9.10E-01	9.19E-01	9.46E-01
CCR: 5	j301_1	1.00E-02					1.04E-01					1.25E-02	1.00E-02				1.91E-02	1.68E-02				2.79E-01	2.75E-01			4.49E-02
	j305_4	2.25E-01				5.34E-02	1.00E-02										1.00E-02					6.20E-01	6.58E-01	6.97E-01	5.10E-01	3.32E-01
	j3010_1	2.20E-01	5.39E-02			1.88E-01	1.32E-01									5.44E-01	3.90E-01				4.39E-01	7.75E-01	6.29E-01	6.12E-01	5.39E-01	6.76E-01
	j3014_4	7.46E-01	1.60E-02	1.60E-02	7.45E-01	1.02E+00	9.38E-01	9.42E-01	2.92E-01		9.45E-01	1.01E+00	1.01E+00	4.69E-01		1.02E+00	1.02E+00		4.90E-01		1.02E+00	9.22E-01	9.25E-01	9.25E-01	9.22E-01	9.26E-01
CCR: 10	j301_1	1.21E-02	1.00E-02				5.91E-02					1.21E-02	1.00E-02				1.57E-02	1.36E-02			1.00E-02	7.08E-02				1.00E-02
	j305_4					1.00E-02	1.28E-02					4.24E-02										4.65E-01	3.49E-01	3.62E-01		3.62E-01
	j3010_1	3.56E-01				1.36E-01						7.05E-01	7.04E-01	7.05E-01	8.64E-01	8.05E-01					3.83E-01	7.51E-01	6.05E-01	6.05E-01	6.30E-01	6.30E-01
	j3014_4	1.03E+00	1.03E+00	1.03E+00	1.03E+00	1.03E+00	1.02E+00	1.02E+00	7.11E-01	7.09E-01	1.02E+00	1.07E+00	1.07E+00	5.56E-01		1.07E+00	1.07E+00	2.73E-01	5.22E-01		1.06E+00	9.89E-01	9.89E-01	9.89E-01	9.87E-01	9.89E-01
CCR: 20	j301_1	1.29E-02	1.00E-02				1.13E-01	8.52E-02				1.29E-02	1.00E-02				1.20E-02	1.00E-02				2.92E-01	2.88E-01	7.42E-02		4.41E-02
	j305_4	3.30E-02				1.33E-02	1.00E-02										1.00E-02						3.76E-01	3.65E-01		3.55E-01
	j3010_1	3.59E-01				1.39E-01	3.67E-01				1.00E-02	7.03E-01	6.99E-01	7.00E-01		8.02E-01	3.11E-01				4.42E-01	8.51E-01	7.29E-01	7.24E-01	8.51E-01	7.96E-01
	j3014_4	1.06E+00	1.06E+00	1.06E+00		1.06E+00	1.07E+00	1.07E+00	5.18E-01		1.07E+00	1.06E+00	1.06E+00	4.78E-01		1.06E+00	1.04E+00	1.04E+00	3.51E-01		1.04E+00	1.05E+00	1.05E+00	1.05E+00	1.05E+00	1.05E+00

Tablica 4.7: Usporedba svih pet inačica algoritma SDSE prema IGD-u

		Platform16c1					Platform16c2					Platform16c3					Platform16c4					Platform16c5				
		SDSE	C-SDSE	CN-SDSE	N-SDSE	MCN-SDSE	SDSE	C-SDSE	CN-SDSE	N-SDSE	MCN-SDSE	SDSE	C-SDSE	CN-SDSE	N-SDSE	MCN-SDSE	SDSE	C-SDSE	CN-SDSE	N-SDSE	MCN-SDSE	SDSE	C-SDSE	CN-SDSE	N-SDSE	MCN-SDSE
CCR: 0.01	j301_1	0.00E+00	2.38E-02				0.00E+00	5.37E-02				9.32E-02	0.00E+00				0.00E+00	2.38E-02				0.00E+00	4.87E-02			2.41E-03
	j305_4											0.00E+00											7.35E-03	3.13E-02	1.62E-01	1.00E-01
	j3010_1	0.00E+00					0.00E+00					1.31E-01	7.96E-02	7.94E-02		5.96E-02	7.32E-02				1.14E-02	3.84E-02	3.29E-03	2.00E-03	1.03E-01	9.71E-04
	j3014_4	0.00E+00	1.13E-01	1.18E-01	1.34E-01	9.32E-02	1.60E-01	6.09E-02			1.97E-01	0.00E+00	3.48E-01	2.53E-01		1.12E-01	0.00E+00	4.19E-01	3.26E-01		1.33E-01	1.01E-01	3.03E-02	2.88E-02	1.56E-01	3.55E-04
CCR: 0.1	j301_1	0.00E+00	2.40E-02				0.00E+00					0.00E+00					0.00E+00	2.38E-02				0.00E+00	2.35E-02			2.16E-02
	j305_4																					2.50E-01	2.79E-02	8.97E-02		7.68E-02
	j3010_1	0.00E+00					0.00E+00					6.02E-01		6.66E-02	4.70E-02	1.61E-02					0.00E+00	2.21E-02	8.00E-02	7.85E-02	1.64E-01	7.79E-02
	j3014_4	1.51E-01	2.09E-01	7.83E-02	1.49E-01	1.93E-01	2.81E-03	6.70E-02	3.57E-01		1.95E-01	1.39E-01	2.24E-01	3.80E-01		7.20E-02	1.33E-01	1.09E-01	3.33E-01		1.75E-01	1.18E-01	6.20E-02	6.23E-02	1.47E-01	6.13E-04
CCR: 0.5	j301_1	0.00E+00					0.00E+00	5.55E-02				0.00E+00	2.42E-02				0.00E+00	2.42E-02				0.00E+00	2.11E-02			1.25E-01
	j305_4	0.00E+00					0.00E+00					0.00E+00					0.00E+00					0.00E+00	1.56E-01	1.44E-01		1.83E-01
	j3010_1	0.00E+00				2.09E-01	0.00E+00				2.37E-01	7.95E-02	6.63E-02	6.48E-02	9.52E-02	7.88E-03	7.28E-02				1.49E-02	6.10E-03	7.16E-02	7.16E-02	3.56E-02	6.52E-02
	j3014_4	7.90E-02	6.96E-02	7.54E-02		2.78E-02	1.15E-01	4.92E-02	2.97E-01		0.00E+00	8.12E-02	1.06E-01	3.33E-01		3.06E-02	1.69E-01	2.08E-01	3.51E-01		0.00E+00	1.08E-01	5.81E-02	5.78E-02	1.76E-01	2.64E-02
CCR: 1	j301_1	0.00E+00	2.55E-02				0.00E+00	5.49E-02				0.00E+00	2.55E-02				0.00E+00	2.55E-02				0.00E+00				2.20E-01
	j305_4	0.00E+00				1.60E-01																0.00E+00	1.75E-01	1.75E-01		1.75E-01
	j3010_1	0.00E+00	3.63E-01			1.67E-01		0.00E+00				2.10E-02	4.27E-02	4.17E-02		0.00E+00	6.82E-02				1.49E-02	1.20E-01	6.51E-02	6.77E-02	7.62E-02	2.00E-01
	j3014_4	7.21E-02	4.63E-02	3.10E-02		1.37E-03	1.55E-01	2.32E-02	4.29E-01		0.00E+00	1.27E-01	1.33E-01	3.53E-01	1.30E-01	0.00E+00	1.53E-01	6.25E-01	3.03E-01		0.00E+00	1.04E-02	3.75E-02	3.75E-02	2.49E-02	0.00E+00
CCR: 5	j301_1	0.00E+00					0.00E+00					0.00E+00	2.49E-02				0.00E+00	2.32E-02				0.00E+00	9.63E-03			5.96E-01
	j305_4	0.00E+00				3.74E-01	0.00E+00										0.00E+00					2.84E-01	5.57E-02	2.60E-02	7.54E-02	5.60E-01
	j3010_1	1.98E-01	3.83E-01			5.25E-02	0.00E+00									0.00E+00	7.41E-02				0.00E+00	0.00E+00	1.03E-01	1.17E-01	2.13E-01	1.30E-01
	j3014_4	2.50E-01	0.00E+00	0.00E+00	2.50E-01	0.00E+00	6.91E-03	2.62E-03	6.17E-01		0.00E+00	1.15E-01	1.15E-01	3.86E-01		0.00E+00	0.00E+00		4.71E-01		2.36E-03	3.49E-03	3.49E-03	3.49E-03	3.49E-03	0.00E+00
CCR: 10	j301_1	0.00E+00	2.14E-02				0.00E+00					0.00E+00	2.14E-02				0.00E+00	2.06E-02			5.67E-02	0.00E+00				6.08E-01
	j305_4					0.00E+00	0.00E+00					0.00E+00										0.00E+00	2.83E-01	2.30E-01		2.30E-01
	j3010_1	0.00E+00				2.84E-01						8.89E-02	9.01E-02	9.03E-02	0.00E+00	7.46E-02					0.00E+00	8.37E-02	1.04E-01	1.19E-01	7.99E-02	1.74E-01
	j3014_4	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	3.01E-03	1.14E-03	2.60E-01	2.60E-01	0.00E+00	1.02E-01	1.02E-01	3.41E-01		0.00E+00	0.00E+00	7.69E-01	4.62E-01		8.49E-04	0.00E+00	1.98E-03	1.98E-03	1.98E-03	0.00E+00
CCR: 20	j301_1	0.00E+00	2.90E-02				0.00E+00	5.96E-02				0.00E+00	2.90E-02				0.00E+00	2.01E-02				0.00E+00	1.02E-02	2.86E-01		6.33E-01
	j305_4	0.00E+00				1.97E-01	0.00E+00										0.00E+00						8.80E-04	2.04E-02		6.64E-02
	j3010_1	0.00E+00				2.74E-01	0.00E+00				7.07E-01	1.34E-01	3.91E-02	3.89E-02		0.00E+00	1.05E-01				0.00E+00	5.56E-04	5.31E-02	7.75E-02	0.00E+00	8.87E-02
	j3014_4	0.00E+00	0.00E+00	0.00E+00		0.00E+00	3.03E-04	3.03E-04	4.66E-01		0.00E+00	1.25E-01	1.25E-01	3.87E-01		0.00E+00	0.00E+00	0.00E+00	5.97E-01		6.14E-04	5.49E-04	5.49E-04	5.49E-04	5.49E-04	0.00E+00

Tablica 4.8: Usporedba svih pet inačica algoritma SDSE prema korelaciji s referentnom frontom

		Platform16c1					Platform16c2					Platform16c3					Platform16c4					Platform16c5				
		SDSE	C-SDSE	CN-SDSE	N-SDSE	MCN-SDSE	SDSE	C-SDSE	CN-SDSE	N-SDSE	MCN-SDSE	SDSE	C-SDSE	CN-SDSE	N-SDSE	MCN-SDSE	SDSE	C-SDSE	CN-SDSE	N-SDSE	MCN-SDSE	SDSE	C-SDSE	CN-SDSE	N-SDSE	MCN-SDSE
CCR: 0.01	j301_1	0.01	0.00				0.01	0.00				0.00	0.01				0.01	0.00				0.01	0.00			0.00
	j305_4											0.01											0.50	0.00	0.00	0.50
	j3010_1	0.01					0.01					0.20	0.20	0.40		0.60	0.67				0.33	0.00	0.00	0.50	0.00	0.50
	j3014_4	0.01	0.00	0.00	0.00	0.00	0.75	0.25			0.00	0.01	0.00	0.00		0.00	0.01	0.00	0.00		0.00	0.00	0.25	0.00	0.00	0.75
CCR: 0.1	j301_1	0.01	0.00				0.01					0.01					0.01	0.00				0.01	0.00			0.00
	j305_4																					0.50	0.50	0.00		0.00
	j3010_1	0.01					0.01					0.00		0.00	0.50	0.50					0.01	0.33	0.33	0.00	0.00	0.33
	j3014_4	0.25	0.00	0.25	0.75	0.00	0.75	0.50	0.00		0.00	0.50	0.25	0.00		85.71	0.75	0.25	0.00		0.00	0.20	0.00	0.00	0.00	0.80
CCR: 0.5	j301_1	0.01					0.01	0.00				0.01	0.00				0.01	0.00				0.01	0.00			0.00
	j305_4	0.01					0.01					0.01					0.01					0.01	0.00	0.00		0.00
	j3010_1	0.01				0.00	0.01				0.00	0.25	0.00	0.00	0.00	0.75	0.33				0.67	0.67	0.00	0.00	0.33	0.00
	j3014_4	0.25	0.25	0.25		0.75	0.00	0.00	0.00		0.01	0.50	0.00	0.00		0.50	0.00	0.00	0.00		0.01	0.50	0.00	0.00	0.00	0.50
CCR: 1	j301_1	0.01	0.00				0.01	0.00				0.01	0.00				0.01	0.00				0.01				0.00
	j305_4	0.01				0.00																0.01	0.00	0.00		0.00
	j3010_1	0.01	0.00			0.00		0.01				0.40	0.00	0.00		0.01	0.33				0.67	0.00	0.40	0.20	0.00	0.40
	j3014_4	0.33	0.33	0.33		0.67	0.00	0.00	0.00		0.01	0.00	0.00	0.00	0.00	0.01	0.00	0.00	0.00		0.01	0.00	0.00	0.00	0.00	0.01
CCR: 0.5	j301_1	0.01					0.01					0.01	0.00				0.01	0.00				0.01	0.00			0.00
	j305_4	0.01				0.01	0.01										0.01					0.14	0.29	0.43	0.43	0.00
	j3010_1	0.33	0.00			0.67	0.01									0.01	0.00				0.01	0.01	0.00	0.00	0.00	0.00
	j3014_4	0.00	0.01	0.01	0.00	0.01	0.00	0.00	0.00		0.01	0.00	0.00	0.00		0.01	0.01		0.00		0.00	0.00	0.00	0.00	0.00	0.01
CCR: 10	j301_1	0.01	0.00				0.01					0.01	0.00				0.01	0.00			0.00	0.01				0.00
	j305_4					0.01	0.01					0.01										0.01	0.00	0.00		0.00
	j3010_1	0.01				0.00						0.00	0.00	0.00	0.01	0.00					0.01	0.50	0.00	0.00	0.50	0.00
	j3014_4	0.01	0.01	0.01	0.01	0.01	0.00	0.00	0.00	0.00	0.01	0.00	0.00	0.00		0.01	0.01	0.00	0.00		0.00	0.01	0.00	0.00	0.00	0.01
CCR: 20	j301_1	0.01	0.00				0.01	0.00				0.01	0.00				0.01	0.00				0.01	0.00	0.00		0.00
	j305_4	0.01				0.00	0.01										0.01						0.75	0.50		0.25
	j3010_1	0.01				0.00	0.01				0.00	0.33	0.00	0.00		0.01	0.00				0.01	0.33	0.00	0.00	0.01	0.00
	j3014_4	0.01	0.01	0.01		0.01	0.00	0.00	0.00		0.01	0.00	0.00	0.00		0.01	0.01	0.01	0.00		0.00	0.00	0.00	0.00	0.00	0.01

Uspješnost pronalazaka mogućih rješenja po pojedinim konfiguracijama platforma je dana u tablici 4.9. Vidljivo je da osnovna inačica algoritma SDSE pronalazi moguća rješenja u najvećem broju slučajeva - u prosjeku je uspješnost 85%. Algoritmi C-SDSE i MCN-SDSE pronalaze moguća rješenja u oko 60% slučajeva, dok su algoritmi CN-SDSE i N-SDSE prilično neuspješni i ne uspijevaju pronaći moguća rješenja niti u polovici slučajeva. Međutim, iz tablica 4.6, 4.7 i 4.8 može se uočiti da za model platforme Platform16c5 algoritmi C-SDSE, CN-SDSE i MCN-SDSE uspijevaju pronaći rješenja tamo gdje SDSE ne može pronaći rješenja. Za ostale modele platforma algoritmi C-SDSE i MCN-SDSE uspijevaju pronaći rješenja u nekim slučajevima gdje SDSE nije pronašao rješenja, no generalno su manje uspješni.

Tablica 4.9: Uspješnost u pronalaženju mogućih rješenja

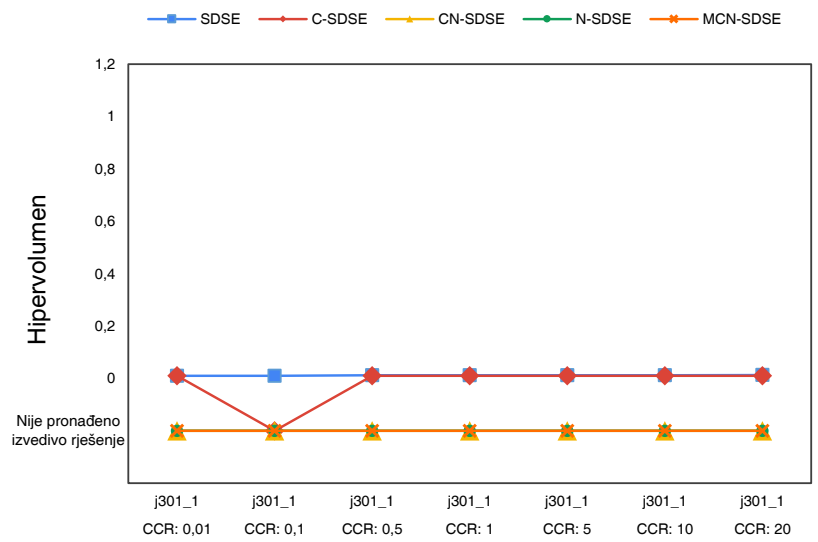
	SDSE	C-SDSE	CN-SDSE	N-SDSE	MCN-SDSE
Platform16c1	89,29%	50,00%	25,00%	14,29%	57,14%
Platform16c2	82,14%	42,86%	21,43%	3,57%	32,14%
Platform16c3	82,14%	64,29%	46,43%	14,29%	50,00%
Platform16c4	78,57%	46,43%	25,00%	0,00%	53,57%
Platform16c5	92,86%	92,86%	78,57%	57,14%	100,00%
Prosjek	85,00%	59,29%	39,29%	17,86%	58,57%

Detaljnija usporedba algoritama prema ostvarenom hipervolumenu je prikazana na slikama 4.31, 4.32 i 4.33. Na slici 4.31 prikazani su reprezentativni slučajevi za platforme c1-c4. Vidljivo je da algoritam SDSE ima najveću uspješnost pri pronalaženju mogućeg rješenja. Algoritam MCN-SDSE u nekim slučajevima uspijeva pronaći rješenja gdje SDSE ne uspijeva, a za aplikacije j3010_1 i j3014_4, u kojima gotovo svi procesori mogu izvesti sve zadaće, algoritam MCN-SDSE postiže najbolje rezultate. Algoritam C-SDSE se za aplikacije j301_1 i j3014_4 ponaša slično kao i SDSE, a u ostalim slučajevima je lošiji. Najlošije performanse pokazuju algoritmi N-SDSE i CN-SDSE koji ili ne uspijevaju uopće pronaći rješenja ili ako ih i pronađu budu znatno lošija od rješenja koja pronađu ostali algoritmi. Rezultati prema mjerama IGD i korelacije s referentnom frontom pokazuju isto ponašanje pa radi sažetosti nisu grafički prikazani.

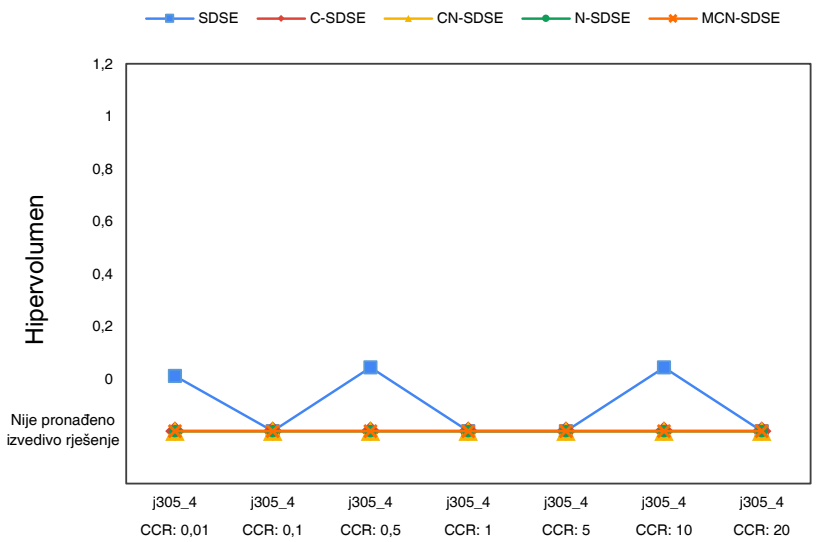
Posebno su obrađeni rezultati za aplikaciju j3014_4 budući da u ovom slučaju svi procesori

mogu izvesti sve zadaće. Rezultati za platforme c1-c4 su prikazani na slici 4.32. U ovom slučaju jedino algoritam N-SDSE ne uspijeva pronaći moguća rješenja u većini slučajeva, dok algoritam MCN-SDSE daje rješenja bolja ili podjednaka algoritmu SDSE. Ostali algoritmi daju uglavnom podjednake ili nešto lošije rezultate od algoritma SDSE. Rezultati prema mjerama IGD i korelacije s referentnom frontom pokazuju isto ponašanje i radi sažetosti nisu grafički prikazani.

Na slici 4.33 prikazani su posebno rezultati za Platform16c5. U slučaju aplikacije j305_4 gdje osnovni algoritam SDSE ne uspijeva u svim slučajevima pronaći rješenja, algoritmi koji se temelje na grozdovima: C-SDSE, CN-SDSE i MCN-SDSE pronalaze rješenja. Generalno, algoritam MCN-SDSE je najstabilniji: jedini pronalazi moguća rješenja u apsolutno svim slučajevima te je po kvaliteti rješenja približno jednak algoritmu SDSE - u nekim slučajevima nešto bolji, u drugima nešto lošiji. Ponovno, najlošije performanse pokazuju algoritmi N-SDSE i CN-SDSE koji ili ne uspijevaju uopće pronaći rješenja ili ako ih i pronađu, ona su lošija od rješenja koja pronađu ostali algoritmi.

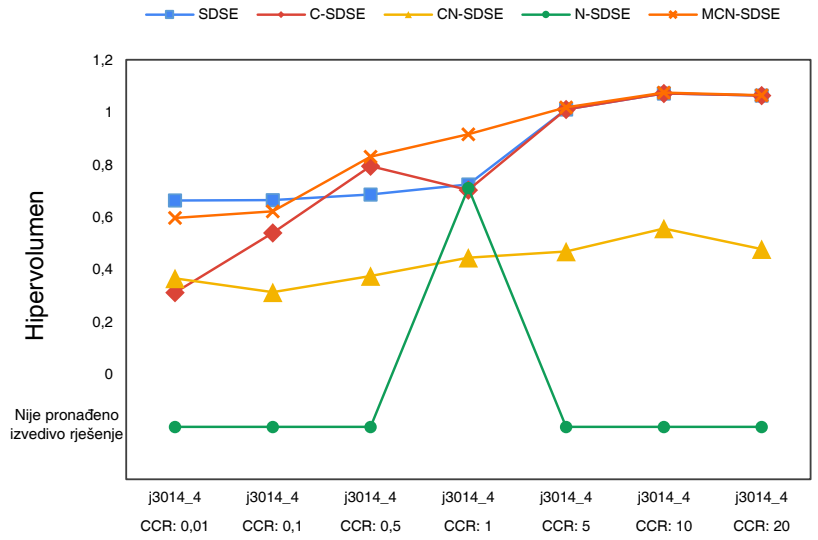


(a) Hipervolumen - j301_1 - Platform16c3



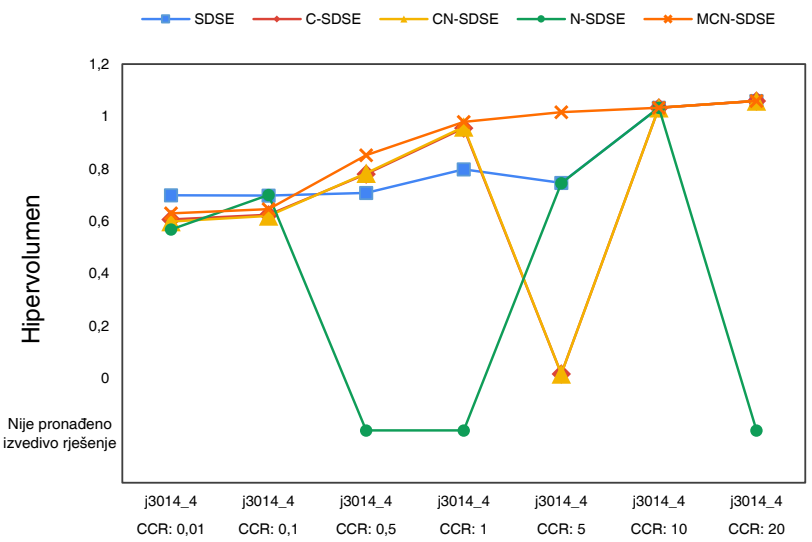
(b) Hipervolumen - j305_4 - Platform16c3

(c) Hipervolumen - j3010_1 - Platform16c4

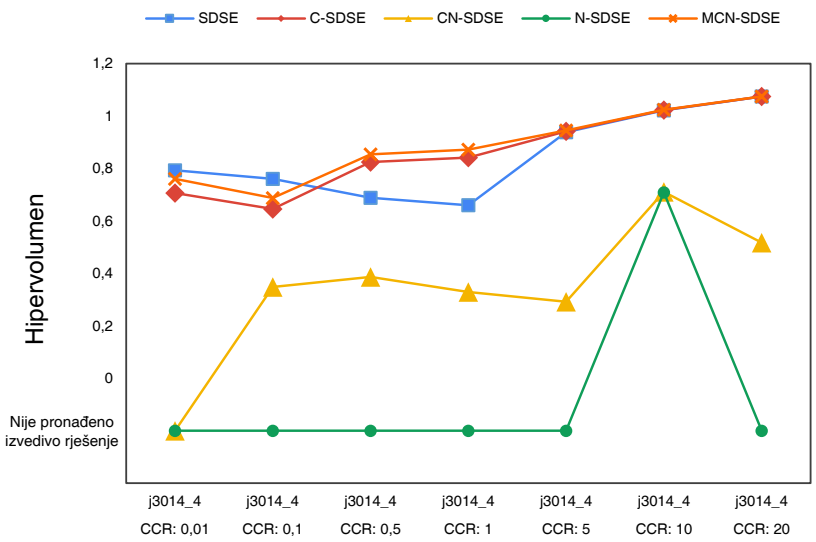


(d) Hipervolumen - j3014_4 - Platform16c3

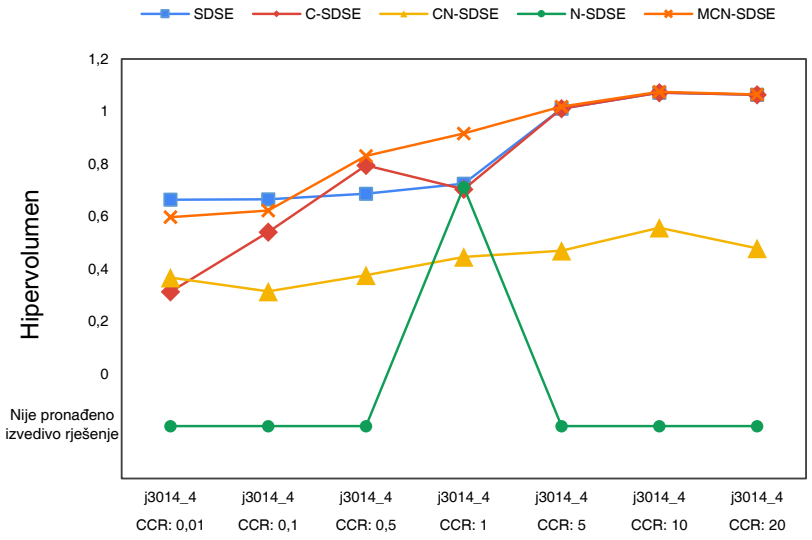
Slika 4.31: Usporedba reprezentativnih slučajeva za sve modele aplikacija na modelima platforma Platform16c1-c4 prema hipervolumenu



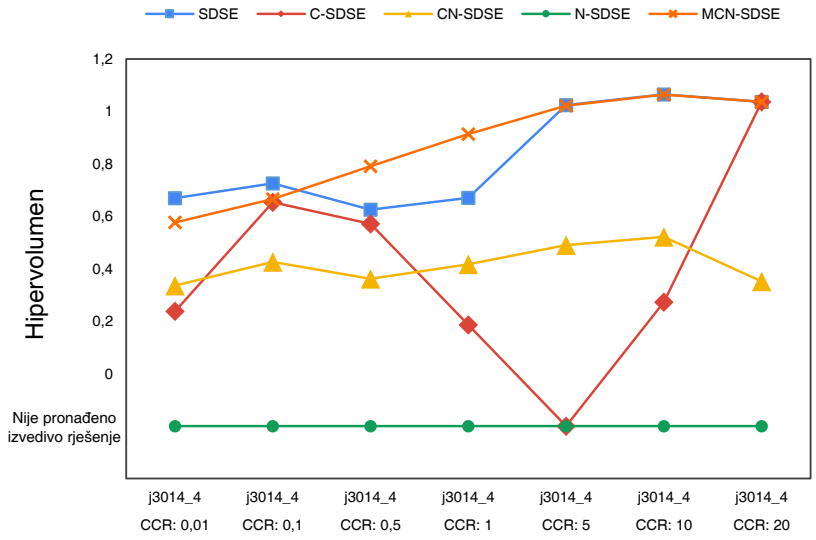
(a) Hipervolumen - j3014_4 - Platform16c1



(b) Hipervolumen - j3014_4 - Platform16c2

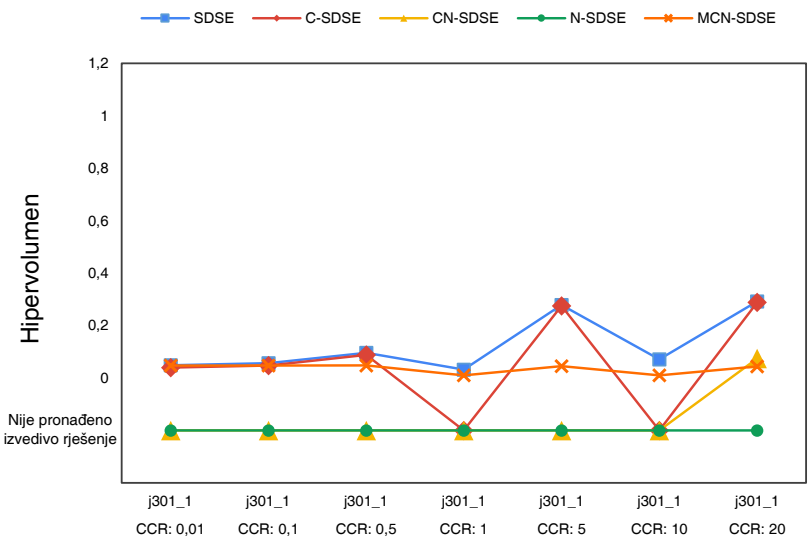


(c) Hipervolumen - j3014_4 - Platform16c3

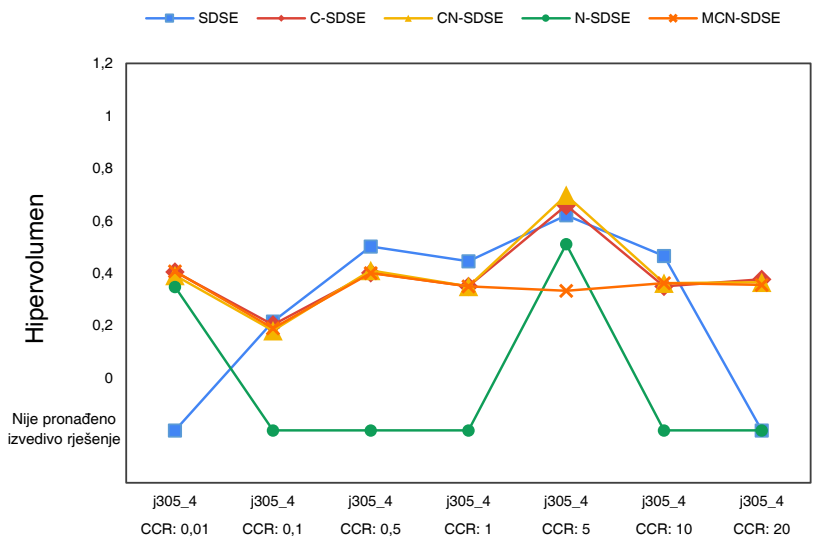


(d) Hipervolumen - j3014_4 - Platform16c4

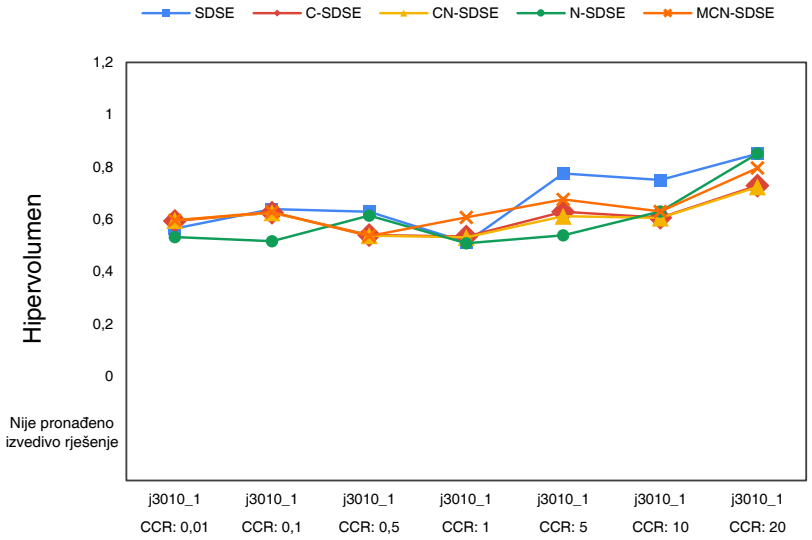
Slika 4.32: Usporedba hipervolumena za aplikaciju j3014_4 na modelima platforma Platform16c1-c4



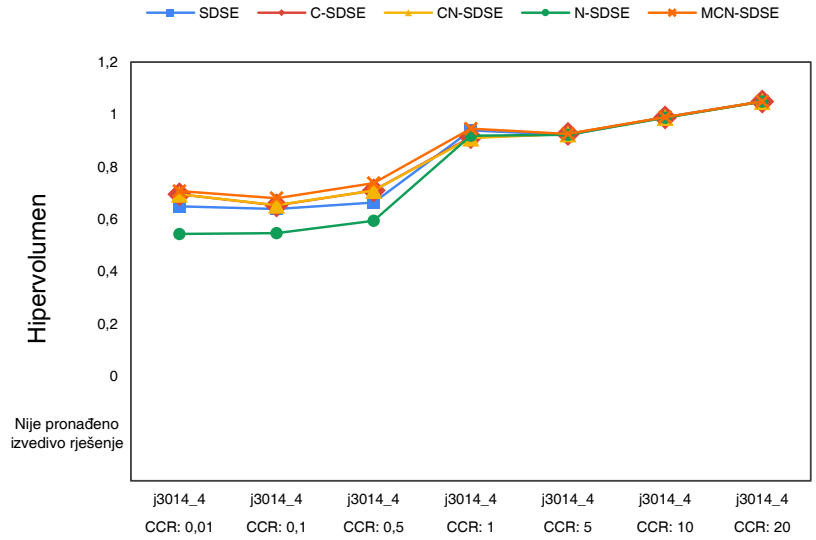
(a) Hipervolumen - j301_1



(b) Hipervolumen - j305_4



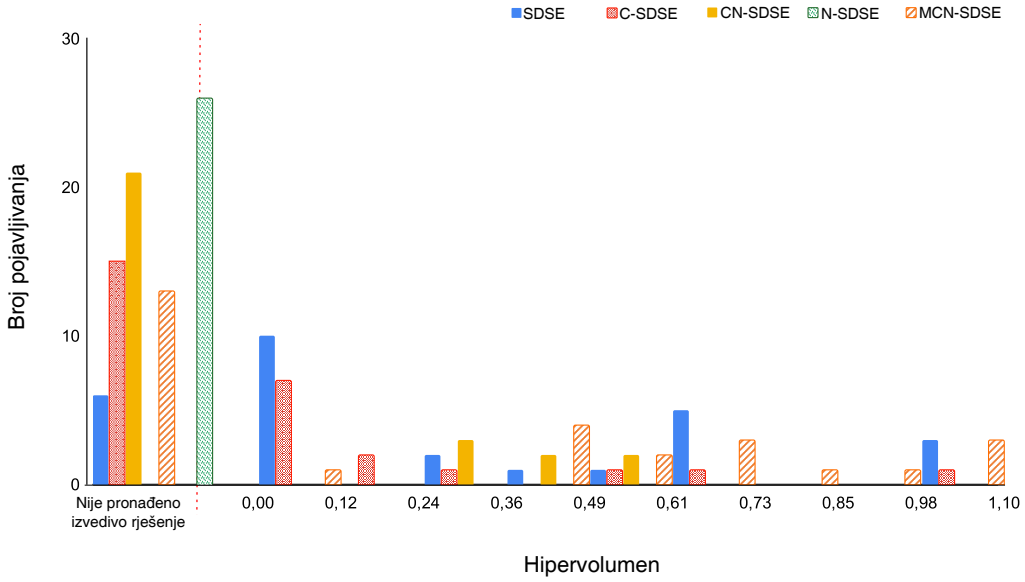
(c) Hipervolumen - j3010_1



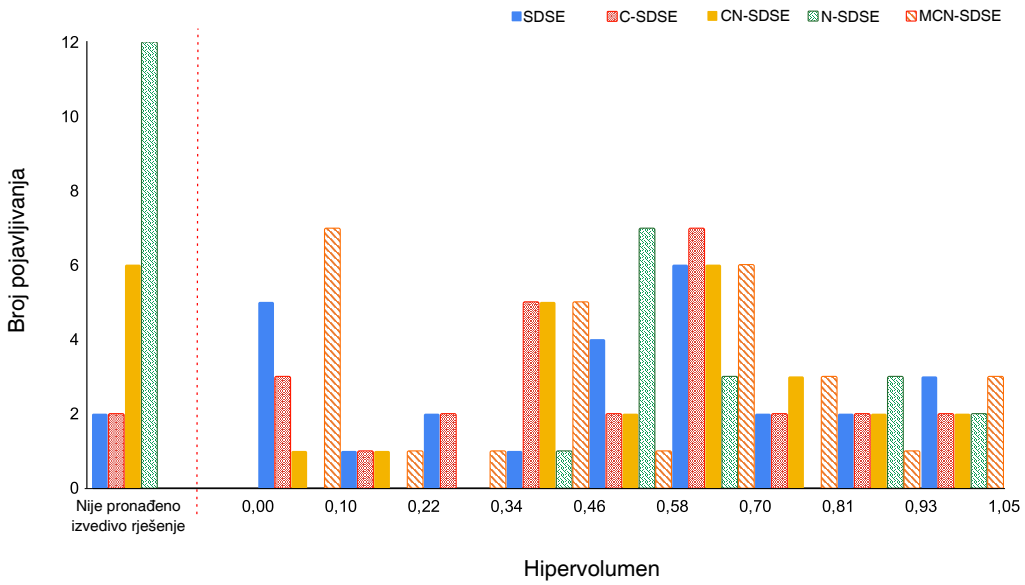
(d) Hipervolumen - j3014_4

Slika 4.33: Usporedba hipervolumena za sve modele aplikacija na platformi Platform16c5

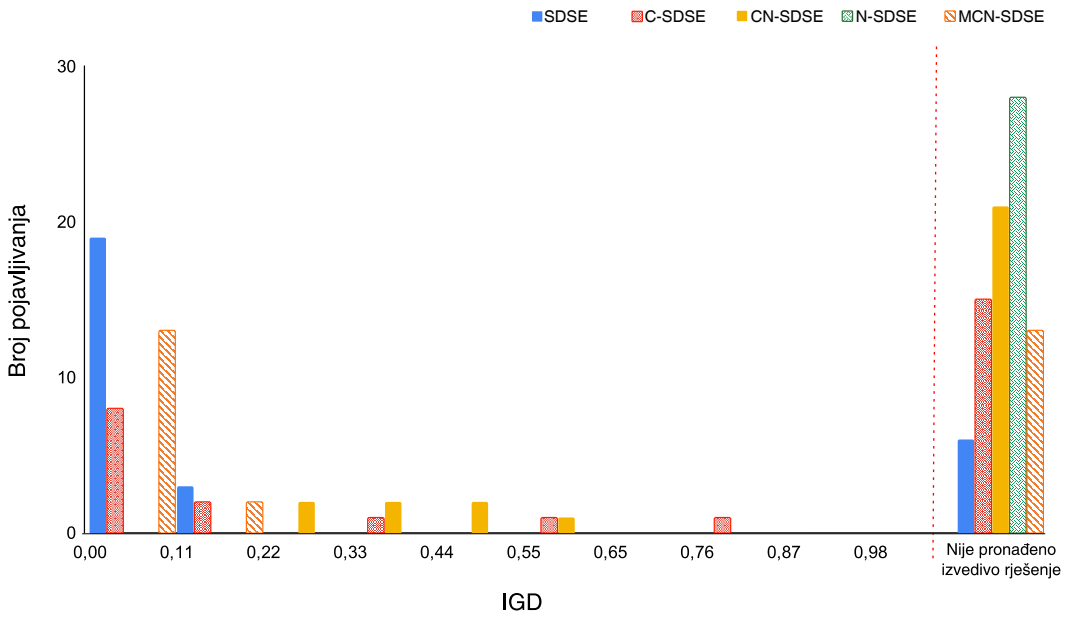
Na slikama 4.34 do 4.39 prikazani su histogrami distribucije postignutih rezultata svih 5 inačica algoritma za sve tri mjere. Rezultati za sve aplikacije i sve faktore CCR su objedinjeni kako bi se lakše sagledala ukupna uspješnost pojedinog algoritma.



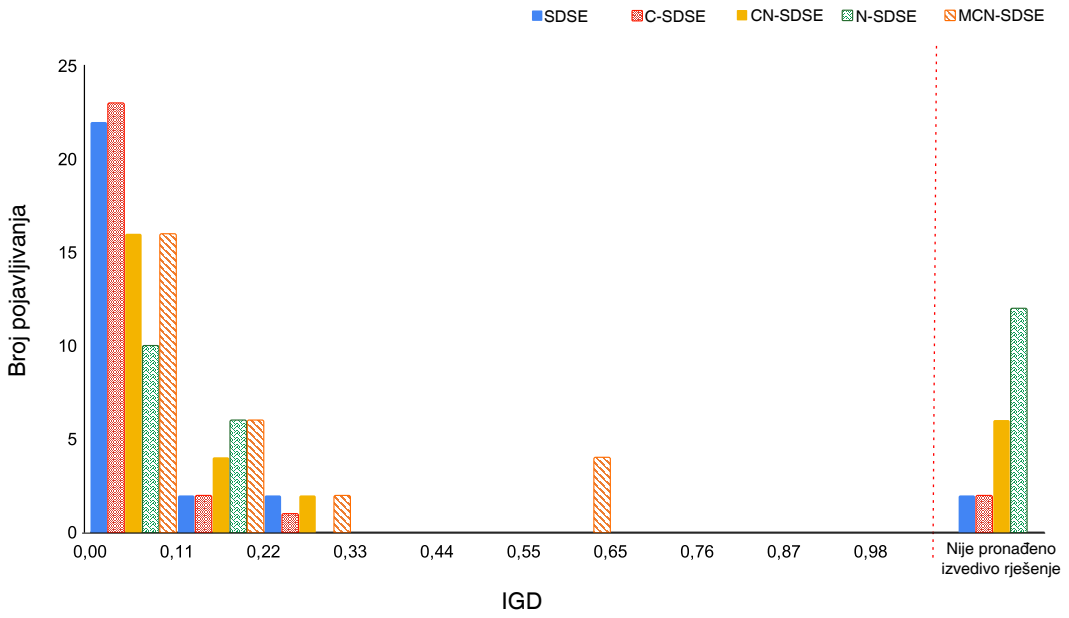
Slika 4.34: Histogram hipervolumena za sve aplikacije i sve faktore CCR na platformi Platform16c4



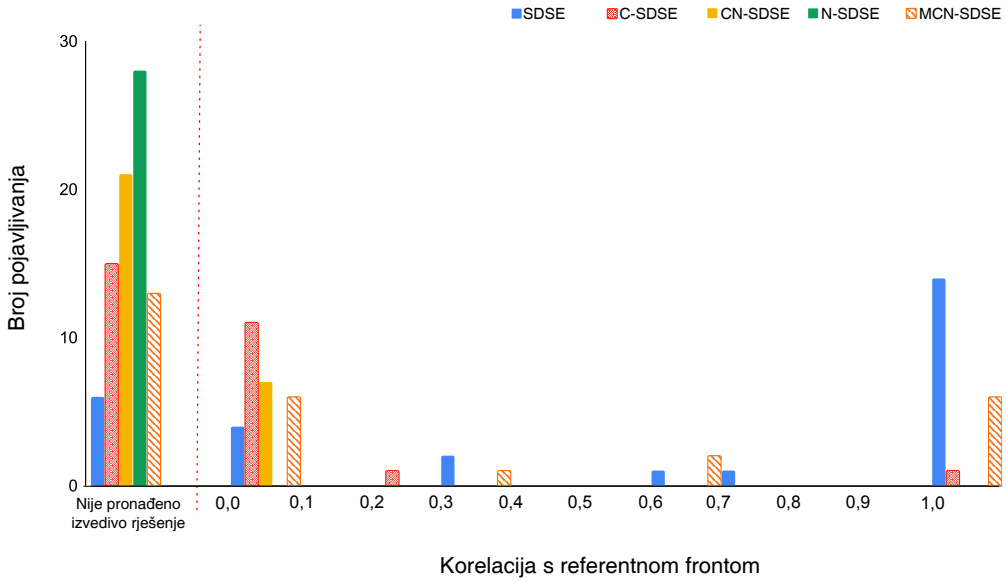
Slika 4.35: Histogram hipervolumena za sve aplikacije i sve faktore CCR na platformi Platform16c5



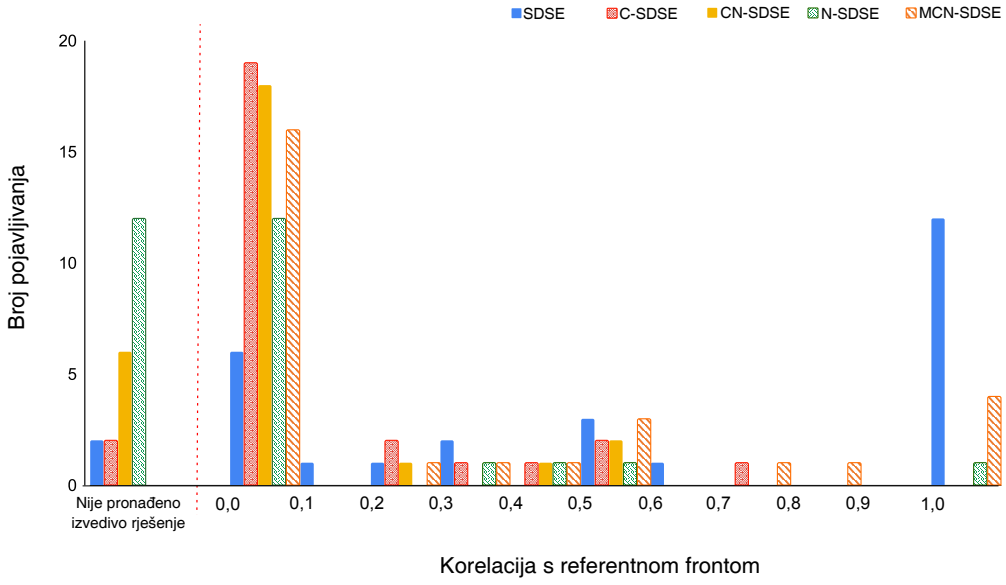
Slika 4.36: Histogram IGD-a za sve aplikacije i sve faktore CCR na platformi Platform16c4



Slika 4.37: Histogram IGD-a za sve aplikacije i sve faktore CCR na platformi Platform16c5



Slika 4.38: Histogram korelacije s referentnom frontom za sve aplikacije i sve faktore CCR na platformi Platform16c4



Slika 4.39: Histogram korelacije s referentnom frontom za sve aplikacije i sve faktore CCR na platformi Platform16c5

Analiza histograma ponovno vodi do jednakih zaključaka. Algoritmi N-SDSE i CN-SDSE su generalno najneuspješniji, mnogo lošiji od algoritma SDSE i nisu pogodni kandidati za primjenu kod rijetko povezanih platforma. Algoritam SDSE je najsvestraniji, tj. pronalazi izvediva i generalno kvalitetna rješenja u velikoj većini slučajeva. Međutim, algoritmi C-SDSE i posebno

MCN-SDSE sudjeluju s većim postotkom među najboljim rješenjima. Zanimljivo je promotriti usporedbu parova algoritama SDSE i N-SDSE te C-SDSE i CN-SDSE. U oba slučaja jedina razlika između dva algoritma iz para je u zamjeni beskonačnog trajanja izvođenja nemogućeg rješenja s penalizacijom nemogućih veza u rješenju koje vodi do gradacije nemogućih rješenja. Međutim, iako bi se intuitivno pomislilo da bi ovakav pristup trebao osigurati veću uspješnost, pokazuje se da izbacivanje beskonačnosti zapravo znatno pogoršava performanse algoritama.

Iz svega izloženoga može se zaključiti da u slučaju pretrage prostora oblikovanja s rijetko povezanim platformama uz algoritam SDSE treba paralelno provesti i algoritam MCN-SDSE. Nadalje, može se uočiti da je algoritam SDSE bolji u slučajevima kada ne mogu svi procesori izvesti sve zadaće, no kada nije takva situacija, algoritam MCN-SDSE koji je bolje prilagođen strukturi platforma iz modela Platform16c1-c5 daje bolje rezulate. Iz toga se može zaključiti da SDSE nije loš generički algoritam i dovoljno je svestran, ali ukoliko se žele ostvariti bolji rezultati za konfiguracije platforme koje su vrlo specifične po pitanju povezanosti treba razmotriti i moguće modifikacije kako bi se prilagodilo problemu.

U budućem istraživanju trebalo bi provjeriti kako bi vraćanje beskonačnog trajanja izvođenja nemogućeg rješenja u algoritam MCN-SDSE utjecalo na performanse algoritma. Također zanimljivo bi bilo ispitati ove algoritme na još raznovrsnijim konfiguracijama rijetko povezanih platforma s nejednakim brojem i vrstom procesora u grozdovima i moguće nešto većim brojem veza između pojedinih procesora.

Poglavlje 5

Zaključak

U ovom radu je predložena metoda za pretragu prostora oblikovanja heterogenih višeprocesorskih platformi. Sustav čine višeprocesorska platforma i aplikacija koja se na njoj izvodi, a modeliraju se na visokoj razini prema načelima SLD-a što podrazumijeva odvajanje ponašanja od arhitekture te izračuna od komunikacije.

Za izradu modela na visokoj razini apstrakcije nužno je bilo napraviti ranu procjenu trajanja izvođenja. U tu svrhu predložena je nova metoda pod nazivom *ELOPS-EM* (*ELementary OPerationS based Estimation Method*) koja daje procjenu trajanja izvođenja na temelju izvornog kôda aplikacije. Metoda ELOPS-EM omogućava identifikaciju zasebnih dijelova izvornog kôda, koji se nazivaju elementarne operacije, a čije se trajanje izvođenja na platformi može mjeriti potpuno neovisno o drugim dijelovima kôda i samoj aplikaciji u kojoj se pojavljuju. Pomoću metode ELOPS-EM moguće je izraditi profile aplikacije i platforme koji su međusobno potpuno nezavisni i koje je moguće ponovno iskoristiti za različite kombinacije aplikacija i platformi. Temeljem tih profila može se dati rana procjena trajanja izvođenja cijele aplikacije bez potrebe za simulacijom na razini osnovnih blokova i bez izrade matematičkog modela podatkovnog puta i priručne memorije procesora.

Metoda ELOPS-EM je evaluirana za algoritme JPEG i AES na nekoliko različitih konfiguracija platforme Xilinx Zynq ZC706. Prosječna pogreška u procjeni trajanja izvođenja iznosila je oko 5%, dok maksimalna pogreška nije prelazila 17%. Postignuta razina točnosti je bolja u usporedbi s analitičkim metodama za oko 350-400%, dok je nešto lošija u usporedbi s točnosti simulacijskih metoda koje imaju razinu maksimalnu pogreške ispod 10%. No, u odnosu na simulacijske metode, metodu ELOPS-EM karakterizira visoka razina ponovne iskoristivosti profila aplikacije i platforme što znači da u slučaju izmjena dijelova kôda aplikacije, pri para-

lelizaciji ili optimizaciji algoritma, nije nužno ponovno raditi vremenski zahtjevna mjerenja na ISS-u ili fizičkoj platformi kao što je slučaj kod metoda koje temelje procjenu na osnovnim blokovima. Mogućnost ponovnog korištenja ranije dobivenih profila pridonosi smanjenju napora i vremena potrebnog za dobivanje rane procjene trajanja izvođenja.

Model sustava nad kojim se radi pretraga prostora oblikovanja obuhvaća modele aplikacije i platforme koji se grade na osnovu profila aplikacija i platforme dobivenih ELOPS-EM metodom. Ti modeli se koriste u heurističkoj metodi pretrage prostora oblikovanja koja ima za cilj pronalaženje optimalne sheme pridruživanja (engl. *mapping scheme*) dijelova aplikacije na elemente platforme te određivanje redoslijeda izvođenja. Pri tome se procedure pridružuje procesorima, a komunikacijske kanale memorijama. Pretraga se temelji na evolucijskom algoritmu za višekriterijsku optimizaciju NSGA-II, a optimiraju se dva kriterija: vrijeme izvođenja i broj elemenata platforme. S obzirom na veliki utjecaj memorijske konfiguracije na performanse cijelog sustava, pretraga prostora oblikovanja obuhvaća i procesore i memorije. Osmišljene su dvije inačice pretrage prostora oblikovanja: istovremena pretraga (SDSE) i dvostupanjska pretraga (2SDSE). U istovremenoj pretrazi, istovremeno se procedure pridružuju procesorima i komunikacijski kanali memorijama. Dvostupanjska pretraga se odvija u dvije iteracije pa se prvo pridružuje procedure procesorima, a zatim komunikacijske kanale memorijama.

Evaluacija je napravljena na umjetno stvorenim modelima i modelima stvarnih aplikacija i platforma, a korištene mjere su bile: hipervolumen, inverzna generacijska udaljenost i korelacija sa referentnom frontom. Ukupno gledano za stvarne i umjetne modele, algoritam SDSE je bez sumnje znatno uspješniji od algoritma 2SDSE prema svim mjerama. U pojedinim slučajevima oba algoritma daju podjednake rezultate, no SDSE ni u jednom slučaju ne daje znatno lošije rezultate od algoritma 2SDSE.

Posebno je razmotren slučaj kada svi procesori nemaju pristup svim memorijama, tj. kada su elementi na platformi rijetko povezani. U takvom slučaju broj nemogućih može znatno premašiti broj mogućih rješenja što problem pretrage prostora oblikovanja čini složenijim. Algoritam SDSE je variran u dodatne 4 inačice posebno prilagođene takvim platformama, a učinkovitost svake inačice je provjerena na 5 posebno stvorenih umjetnih modela. Osnovna inačica algoritma SDSE se pokazala kao najsvestranija, tj. pronašla je izvediva i generalno kvalitetna rješenja u velikoj većini slučajeva. Međutim, inačice C-SDSE i posebno MCN-SDSE sudjeluju s većim postotkom među najboljim rješenjima. Razlog je u tome što su ove inačice bolje prilagođene grozdastoj strukturi rijetko povezanih ispitnih platformi jer detektiraju skupine procesora

povezanih na istu memoriju - grozdove i najprije rade pridruživanje na grozdove, a zatim na procesore u grozdovima. Također se pokazuje da zamjena beskonačnog trajanja izvođenja nemogućeg rješenja s penalizacijom nemogućih veza u rješenju koje vodi do gradacije nemogućih rješenja ne poboljšava performanse algoritama. Zaključak je da SDSE nije loš generički algoritam i dovoljno je svestran, ali ukoliko se žele ostvariti bolji rezultati za konfiguracije platforme koje su vrlo specifične po pitanju povezanosti treba razmotriti i moguće modifikacije kako bi se prilagodilo problemu.

U budućem istraživanju potrebno je u metodu ELOPS-EM ugraditi automatiziranu instrumentaciju kôda za dijelove aplikacije čije je trajanje izvođenja podatkovno ovisno te provjeriti kako se koncept ponaša za procesore s arhitekturom DSP. Nadalje, metoda ELOPS-EM je ispitana na aplikacijama tipičnim za ugradbene sustave koje nemaju velike zahtjeve za priručnom memorijom i stoga bi trebalo ispitati kolika je točnost procjene za aplikacije koje rade s većom količinom podataka i kod kojih je veća vjerojatnost promašaja priručne memorije. Za samu metodu pretrage prostora oblikovanja zanimljivo bi bilo ispitati performanse na još raznovrsnijim konfiguracijama rijetko povezanih platforma s nejednakim brojem i vrstom procesora u grozdovima i moguće nešto većim brojem veza između pojedinih procesora.

Literatura

[1] International Technology Roadmap for Semiconductors, “System Drivers Abstract”, http://www.semiconductors.org/clientuploads/Research{_}Technology/ITRS/2013/2013SysDrivers{_}Summary.pdf, accessed: 2016-12-04. 2013.

[2] Gajski, D. D., Vahid, F., “Specification and design of embedded hardware-software systems”, IEEE Design and Test of Computers, Vol. 12, No. 1, Spring 1995, str. 53-67.

[3] Zhao, Z., Gerstlauer, A., John, L. K., “Source-level performance, energy, reliability, power and thermal (perpt) simulation”, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 36, No. 2, Feb 2017, str. 299-312.

[4] Lin, K. L., Lo, C. K., Tsay, R. S., “Source-level timing annotation for fast and accurate TLM computation model generation”, in Proceedings of the Asia and South Pacific Design Automation Conference, ASP-DAC. IEEE, jan 2010, str. 235–240, dostupno na: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5419890

[5] Wang, Z., Henkel, J., “Accurate source-level simulation of embedded software with respect to compiler optimizations”, Design, Automation & Test in Europe Conference & Exhibition, Vol. 0, 2012, str. 382–387.

[6] Cheung, E., Hsieh, H., Balarin, F., “Fast and accurate performance simulation of embedded software for MPSoC”, in Proceedings of the Asia and South Pacific Design Automation Conference, ASP-DAC. IEEE, jan 2009, str. 552–557, dostupno na: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4796538

[7] Gerin, P., Hamayun, M. M., Pétrot, F., “Native MPSoC co-simulation environment for software performance estimation”, in Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis - CODES+ISSS

’09. New York, New York, USA: ACM Press, 2009, str. 403, dostupno na: http://dl.acm.org/citation.cfm?id=1629435.1629490

[8] Xilinx, “Ug954 - zc706 evaluation board for the zynq-7000 xc7z045 soc user guide (v1.7)”, https://www.xilinx.com/support/documentation/boards_and_kits/zc706/ug954-zc706-eval-board-xc7z045-ap-soc.pdf, 2018.

[9] Adapteva, “Parallella-1.x reference manual”, http://www.parallella.org/docs/parallella_manual.pdf, 2014.

[10] Wallace, G. K., “The JPEG still picture compression standard”, Communications of the ACM, Vol. 34, No. 4, apr 1991, str. 30–44, dostupno na: http://portal.acm.org/citation.cfm?doid=103085.103089

[11] NIST, “Advanced encryption standard (aes) - fips 197”, http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf, accessed: 2016-12-10. 2001.

[12] Deb, K., Pratap, A., Agarwal, S., Meyarivan, T., “A fast and elitist multiobjective genetic algorithm: Nsga-ii”, IEEE Transactions on Evolutionary Computation, Vol. 6, No. 2, April 2002, str. 182-197.

[13] Gajski, D. D., Abdi, S., Gerstlauer, A., Schirner, G., Embedded System Design. Boston, MA: Springer US, 2009, dostupno na: http://link.springer.com/10.1007/978-1-4419-0504-8

[14] Marwedel, P., Embedded System Design. Dordrecht: Springer Netherlands, 2011, dostupno na: http://link.springer.com/10.1007/978-94-007-0257-8

[15] Popovici, K., Guerin, X., Rousseau, F., Paolucci, P. S., Jerraya, A. A., “Platform-based software design flow for heterogeneous MPSoC”, ACM Transactions on Embedded Computing Systems, Vol. 7, No. 4, jul 2008, str. 1–23, dostupno na: http://doi.acm.org/10.1145/1376804.1376807http://portal.acm.org/citation.cfm?doid=1376804.1376807

[16] Wolf, W., “Hardware and Software Co-design”, in High-Performance Embedded Computing. Elsevier, 2007, str. 383–432, dostupno na: http://linkinghub.elsevier.com/retrieve/pii/B9780123694850500087

[17] Zhang, J., Schirner, G., “Towards closing the specification gap by integrating algorithm-level and system-level design”, Design Automation for Embedded Systems, Vol. 19, No. 4, Dec 2015, str. 389–419, dostupno na: https://doi.org/10.1007/s10617-015-9161-1

[18] Nuzzo, P., “From electronic design automation to cyber-physical system design automation: A tale of platforms and contracts”, in Proceedings of the 2019 International Symposium on Physical Design, ser. ISPD ’19. New York, NY, USA: ACM, 2019, str. 117–121, dostupno na: http://doi.acm.org/10.1145/3299902.3311070

[19] Kienhuis, B., Deprettere, E., Vissers, K., Wolf, P. V. D., “An approach for quantitative analysis of application-specific dataflow architectures”, in Proceedings IEEE International Conference on Application-Specific Systems, Architectures and Processors, July 1997, str. 338-349.

[20] Vivekanandarajah, K., Pilakkat, S. K., “Task Mapping in Heterogeneous MPSoCs for System Level Design”, in 13th IEEE International Conference on Engineering of Complex Computer Systems (iceccs 2008). IEEE, mar 2008, str. 56–65, dostupno na: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4492879http://ieeexplore.ieee.org/document/4492879/

[21] van der Putten, P., Voeten, J., Geilen, M., Stevens, M., “System level design methodology”, in Proceedings IEEE Computer Society Workshop on VLSI’98 System Level Design (Cat. No.98EX158). IEEE Comput. Soc, str. 11–16, dostupno na: http://ieeexplore.ieee.org/document/667107/

[22] Keutzer, K., Newton, A., Rabaey, J., Sangiovanni-Vincentelli, A., “System-level design: orthogonalization of concerns and platform-based design”, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 19, No. 12, dec 2000, str. 1523–1543, dostupno na: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=898830http://ieeexplore.ieee.org/document/898830/

[23] Sangiovanni-Vincentelli, A., “Quo Vadis, SLD? Reasoning About the Trends and Challenges of System Level Design”, Proceedings of the IEEE, Vol. 95, No. 3, mar 2007, str. 467–506, dostupno na: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4167779http://ieeexplore.ieee.org/document/4167779/

[24] Chakravarty, S., Zhao, Z., Gerstlauer, A., "Automated, retargetable back-annotation for host compiled performance and power modeling", in 2013 International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2013. IEEE, sep 2013, str. 1–10, dostupno na: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6659023

[25] Oyamada, M., Wagner, F. R., Bonaciu, M., Cesario, W., Jerraya, A., "Software Performance Estimation in MPSoC Design", in 2007 Asia and South Pacific Design Automation Conference. IEEE, jan 2007, str. 38–43, dostupno na: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4195993

[26] Palermo, G., Silvano, C., Zaccaria, V., "ReSPIR: A Response Surface-Based Pareto Iterative Refinement for Application-Specific Design Space Exploration", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 28, No. 12, dec 2009, str. 1816–1829, dostupno na: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5324029

[27] Cilardo, A., Gallo, L., Mazzocca, N., "Design space exploration for high-level synthesis of multi-threaded applications", Journal of Systems Architecture, Vol. 59, No. 10, nov 2013, str. 1171–1183, dostupno na: http://linkinghub.elsevier.com/retrieve/pii/S1383762113001537

[28] Erbas, C., Pimentel, A. D., Thompson, M., Polstra, S., "A Framework for System-Level Modeling and Simulation of Embedded Systems Architectures", EURASIP Journal on Embedded Systems, Vol. 2007, 2007, str. 1–11, dostupno na: http://jes.eurasipjournals.com/content/2007/1/082123

[29] Nikolov, H., Thompson, M., Stefanov, T., Pimentel, A., Polstra, S., Bose, R., Zissulescu, C., Deprettere, E., "Daedalus: Toward composable multimedia mp-soc design", in 2008 45th ACM/IEEE Design Automation Conference, June 2008, str. 574-579.

[30] Leupers, R., Castrillon, J., "MPSoC programming using the MAPS compiler", in 2010 15th Asia and South Pacific Design Automation Conference (ASP-DAC). Taipei: IEEE, jan 2010, str. 897–902, dostupno na: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5419677

[31] Gao, L., Huang, J., Ceng, J., Leupers, R., Ascheid, G., Meyr, H., "TotalProf", in Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis - CODES+ISSS '09. New York, New York, USA: ACM Press, 2009, str. 305, dostupno na: http://portal.acm.org/citation.cfm?doid=1629435.1629477

[32] Abdi, S., Schirner, G., Hwang, Y., Gajski, D. D., Yu, L., "Automatic TLM generation for early validation of multicore sSystems", IEEE Design and Test of Computers, Vol. 28, No. 3, may 2011, str. 10–19, dostupno na: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5620889

[33] Thiele, L., Chakraborty, S., Gries, M., Künzli, S., "A Framework for Evaluating Design Tradeoffs in Packet Processing Architectures", in Proceedings of the 39th Annual Design Automation Conference, ser. DAC '02. New York, NY, USA: ACM, 2002, str. 880–885, dostupno na: http://doi.acm.org/10.1145/513918.514136

[34] Wandeler, E., Thiele, L., Verhoef, M., Lieverse, P., "System architecture evaluation using modular performance analysis: a case study", International Journal on Software Tools for Technology Transfer, Vol. 8, No. 6, oct 2006, str. 649–667, dostupno na: http://dx.doi.org/10.1007/s10009-006-0019-5http://link.springer.com/10.1007/s10009-006-0019-5

[35] Javaid, H., Ignjatovic, A., Parameswaran, S., "Performance Estimation of Pipelined MultiProcessor System-on-Chips (MPSoCs)", IEEE Transactions on Parallel and Distributed Systems, Vol. 25, No. 8, aug 2014, str. 2159–2168, dostupno na: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6636892

[36] Roloff, S., Hannig, F., Teich, J., "Fast architecture evaluation of heterogeneous MPSoCs by host-compiled simulation", in Proceedings of the 15th International Workshop on Software and Compilers for Embedded Systems - SCOPES '12. New York, New York, USA: ACM Press, 2012, str. 52–61, dostupno na: http://dl.acm.org/citation.cfm?doid=2236576.2236582

[37] Flasskamp, M., Sievers, G., Ax, J., Klarhorst, C., Jungeblut, T., Kelly, W., Thies, M., Porrmann, M., "Performance estimation of streaming applications for hierarchical MPSoCs", in Proceedings of the 2016 Workshop on Rapid Simulation and Performance

Evaluation Methods and Tools - RAPIDO '16. New York, New York, USA: ACM Press, 2016, str. 1–6, dostupno na: http://dl.acm.org/citation.cfm?doid=2852339.2852342

[38] Altenbernd, P., Gustafsson, J., Lisper, B., Stappert, F., "Early execution time-estimation through automatically generated timing models", Real-Time Systems, Vol. 52, No. 6, nov 2016, str. 731–760, dostupno na: http://link.springer.com/10.1007/s11241-016-9250-7

[39] Standards, O., "Iso/iec 9899:2011", http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1570.pdf, 2011.

[40] Ivošević, D., Sruk, V., "Unified flow of custom processor design and fpga implementation", in Eurocon 2013, July 2013, str. 1721-1727.

[41] Ivošević, D., Frid, N., Sruk, V., "Function-level performance estimation for heterogeneous mpsoc platforms", in 2016 Zooming Innovation in Consumer Electronics International Conference (ZINC), June 2016, str. 38-41.

[42] Xilinx, "Ug984 - microblaze processor reference guide (v2014.1)", https://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_2/ug984-vivado-microblaze-ref.pdf, 2008.

[43] ARM, "Cortex™-a9 mpcore® technical reference manual, revision: r3p0", http://infocenter.arm.com/help/topic/com.arm.doc.ddi0407g/DDI0407G_cortex_a9_mpcore_r3p0_trm.pdf, 2008.

[44] Bui, B. D., Caccamo, M., Sha, L., Martinez, J., "Impact of cache partitioning on multi-tasking real time embedded systems", in 2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, Aug 2008, str. 101-110.

[45] Frid, N., Ivošević, D., Sruk, V., "Elementary operations: a novel concept for source-level timing estimation", Automatika, Vol. 60, No. 1, 2019, str. 91-104, dostupno na: https://doi.org/10.1080/00051144.2019.1581695

[46] Zitzler, E., Laumanns, M., Thiele, L., "SPEA2: Improving the Strength Pareto Evolutionary Algorithm", in Evolutionary Methods for Design Optimization and Control with Applications to Industrial Problems. Barcelona: International Center for Numerical Methods in Engineering (CIMNE), 2001, str. 95–100.

[47] Castrillon, J., Velasquez, R., Stulova, A., Weihua Sheng, Jianjiang Ceng, Leupers, R., Ascheid, G., Meyr, H., "Trace-based KPN composability analysis for mapping simultaneous applications to MPSoC platforms", in 2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010). Dresden: IEEE, mar 2010, str. 753–758, dostupno na: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5456950

[48] Lin, J., Srivatsa, A., Gerstlauer, A., Evans, B. L., "Heterogeneous multiprocessor mapping for real-time streaming systems", in 2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), May 2011, str. 1605-1608.

[49] Goens, A., Castrillon, J., Odendahl, M., Leupers, R., "An optimal allocation of memory buffers for complex multicore platforms", Journal of Systems Architecture, Vol. 66-67, 2016, str. 69 - 83, dostupno na: http://www.sciencedirect.com/science/article/pii/S1383762116300352

[50] Verma, M., Wehmeyer, L., Marwedel, P., "Cache-aware scratchpad-allocation algorithms for energy-constrained embedded systems", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 25, No. 10, Oct 2006, str. 2035-2051.

[51] Lee, C., Potkonjak, M., Mangione-Smith, W. H., "Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems", in Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture, ser. MICRO 30. Washington, DC, USA: IEEE Computer Society, 1997, str. 330–335, dostupno na: http://dl.acm.org/citation.cfm?id=266800.266832

[52] Falk, H., Kleinsorge, J. C., "Optimal static WCET-aware scratchpad allocation of program code", in Proceedings of the 46th Annual Design Automation Conference on ZZZ - DAC '09. ACM Press, 2009, dostupno na: https://doi.org/10.1145%2F1629911.1630101

[53] Chen, T., Roychoudhury, A., Mitra, T., Suhendra, V., "Wcet centric data allocation to scratchpad memory", in 26th IEEE International Real-Time Systems Symposium (RTSS'05)(RTSS), Vol. 00, 12 2005, str. 223-232, dostupno na: doi.ieeecomputersociety.org/10.1109/RTSS.2005.45

[54] Ozturk, O., Chen, G., Kandemir, M., Karakoy, M., "An integer linear programming based approach to simultaneous memory space partitioning and data allocation for chip multipro-

cessors", in IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures (ISVLSI'06), March 2006, str. 6 pp.-.

[55] Salamy, H., Ramanujam, J., "An effective solution to task scheduling and memory partitioning for multiprocessor system-on-chip", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 31, No. 5, May 2012, str. 717-725.

[56] Jovanovic, O., Kneuper, N., Engel, M., Marwedel, P., "Ilp-based memory-aware mapping optimization for mpsocs", in 2012 IEEE 15th International Conference on Computational Science and Engineering, Dec 2012, str. 413-420.

[57] Jantsch, A., Modeling Embedded Systems and SoC's. Elsevier, 2003, dostupno na: http://www.sciencedirect.com/science/article/pii/B9781558609259500111

[58] Gajski, D. D., Principles of Digital Design. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996.

[59] Kwok, Y.-K., Ahmad, I., "Benchmarking and Comparison of the Task Graph Scheduling Algorithms", Journal of Parallel and Distributed Computing, Vol. 59, No. 3, dec 1999, str. 381–422, dostupno na: http://linkinghub.elsevier.com/retrieve/pii/S0743731599915782

[60] Branke, J., Deb, K., Miettinen, K., Słowiński, R., Multiobjective optimization. Interactive and evolutionary approaches, 01 2008, Vol. 5252.

[61] Zitzler, E., Laumanns, M., Bleuler, S., "A tutorial on evolutionary multiobjective optimization", in Metaheuristics for Multiobjective Optimisation, Gandibleux, X., Sevaux, M., Sörensen, K., T'kindt, V., (ur.). Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, str. 3–37.

[62] Hadka, D., "MOEA Framework", dostupno na: http://moeaframework.org/

[63] Hadka, D., Beginner's Guide to the MOEA Framework, 2017.

[64] Kolisch, R., Sprecher, A., "Psplib - a project scheduling problem library: Or software - orsep operations research software exchange program", European Journal of Operational Research, Vol. 96, No. 1, 1997, str. 205 - 216, dostupno na: http://www.sciencedirect.com/science/article/pii/S0377221796001701

[65] Zitzler, E., Thiele, L., Laumanns, M., Fonseca, C. M., da Fonseca, V. G., "Performance assessment of multiobjective optimizers: An analysis and review", Trans. Evol. Comp, Vol. 7, No. 2, Apr. 2003, str. 117–132, dostupno na: http://dx.doi.org/10.1109/TEVC.2003.810758

[66] Bosman, P. A. N., Thierens, D., "The balance between proximity and diversity in multiobjective evolutionary algorithms", IEEE Transactions on Evolutionary Computation, Vol. 7, No. 2, April 2003, str. 174-188.

[67] Zitzler, E., Laumanns, M., Thiele, L., Fonseca, C. M., da Fonseca, V. G., "Why quality assessment of multiobjective optimizers is difficult", in Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation, ser. GECCO'02. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002, str. 666–674, dostupno na: http://dl.acm.org/citation.cfm?id=2955491.2955590

[68] Knowles, J., Corne, D., "On metrics for comparing nondominated sets", in Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No.02TH8600), Vol. 1, May 2002, str. 711-716 vol.1.

[69] Coello, C. C., Lamont, G. B., van Veldhuizen, D. A., Evolutionary Algorithms for Solving Multi-Objective Problems. Springer US, 2007.

[70] Frid, N., Sruk, V., "Memory-aware multiobjective design space exploration of heteregeneous mpsoc", in 2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), May 2018, str. 0861-0866.

Kazalo

Popis slika

Popis tablica

Dodatak A

Profil platforme za konfiguraciju ARM1 - O0

```
<platform name="ARM1" optlvl="0">
    <operation name="INT_loc_var ADD">
        <single>
            <base>1,351500e-08</base>
            <mod type="const">1,501470e-08</mod>
        </single>
        <seqop nr="2">
            <base>1,351500e-08</base>
        </seqop>
        <seqop nr="3">
            <base>1,670250e-083</base>
        </seqop>
        <seqop nr="5">
            <base>2,139000e-08</base>
        </seqop>
        <seqop nr="10">
            <base>3,699000e-08</base>
        </seqop>
        <seqstat nr="2">
            <base> 2,251500e-08</base>
            <mod type="const">1,501710e-08</mod>
        </seqstat>
        <seqstat nr="3">
            <base>2,851470e-08</base>
            <mod type="const">1,651500e-08</mod>
        </seqstat>
        <seqstat nr="5">
            <base>3,938970e-08</base>
            <mod type="const">2,401470e-08</mod>
        </seqstat>
        <seqstat nr="10">
            <base>5,829000e-08</base>
            <mod type="const">4,715710e-08</mod>
```

```
34          </seqstat>
35      </operation>
36      <operation name="INT_loc_arr ADD" type="simple" dim="1">
37          <single>
38              <base>5,139000e-08</base>
39              <mod type="var">3,900000e-08</mod>
40              <mod type="const">3,753000e-08</mod>
41          </single>
42          <seqop nr="2">
43              <base>6,456000e-08</base>
44          </seqop>
45          <seqop nr="3">
46              <base>7,983000e-08</base>
47          </seqop>
48          <seqstat nr="2">
49              <base>9,426000e-08</base>
50              <mod type="var">7,392000e-08</mod>
51              <mod type="const">6,453000e-08</mod>
52          </seqstat>
53          <seqstat nr="3">
54              <base>1,338600e-07</base>
55              <mod type="var">9,729000e-08</mod>
56              <mod type="const">9,282000e-08</mod>
57          </seqstat>
58          <seqstat nr="5">
59              <base>2,159400e-07</base>
60              <mod type="var">1,608900e-07</mod>
61              <mod type="const">1,491000e-07</mod>
62          </seqstat>
63          <seqstat nr="10">
64              <base>4,228800e-07</base>
65              <mod type="var">3,017800e-07</mod>
66              <mod type="const">2,786000e-07</mod>
67          </seqstat>
68      </operation>
69      <operation name="INT_loc_arr ADD" type="simple" dim="2">
70          <single>
71              <base>8,148000e-08</base>
72              <mod type="var"> 6,207000e-08</mod>
73              <mod type="const">5,994000e-08</mod>
74          </single>
75          <seqop nr="2">
76              <base>1,590096e-07</base>
77          </seqop>
78          <seqop nr="3">
79              <base>1,265723e-07</base>
80          </seqop>
```

```
81          <seqstat nr="2">
82              <base>1,605100e-07</base>
83              <mod type="var">1,222780e-07</mod>
84              <mod type="const">1,180811e-07</mod>
85          </seqstat>
86          <seqstat nr="3">
87              <base>2,363730e-07</base>
88              <mod type="var">1,800650e-07</mod>
89              <mod type="const">1,738900e-07</mod>
90          </seqstat>
91          <seqstat nr="5">
92              <base>3,852437e-07</base>
93              <mod type="var">2,934670e-07</mod>
94              <mod type="const">2,834000e-07</mod>
95          </seqstat>
96          <seqstat nr="10">
97              <base>7,406532e-07</base>
98              <mod type="var">5,642163e-07</mod>
99              <mod type="const">5,448600e-07</mod>
100         </seqstat>
101     </operation>
102     <operation name="INT_loc_arr ADD" type="simple" dim="3">
103         <single>
104             <base>6,951900e-08</base>
105             <mod type="var">6,078900e-08</mod>
106             <mod type="const">5,050500e-08</mod>
107         </single>
108         <seqop nr="2">
109             <base>1,356675e-07</base>
110         </seqop>
111         <seqop nr="3">
112             <base>1,079920e-07</base>
113         </seqop>
114         <seqstat nr="2">
115             <base>1,369524e-07</base>
116             <mod type="var">1,197543e-07</mod>
117             <mod type="const">9,949490e-08</mod>
118         </seqstat>
119         <seqstat nr="3">
120             <base>2,016800e-07</base>
121             <mod type="var">1,756659e-07</mod>
122             <mod type="const">1,456200e-07</mod>
123         </seqstat>
124         <seqstat nr="5">
125             <base>3,286800e-07</base>
126             <mod type="var">2,584320e-07</mod>
127             <mod type="const">2,068500e-07</mod>
```

```
128             </seqstat>
129             <seqstat nr="10">
130                 <base>6,321540e-07</base>
131                 <mod type="var">5,059440e-07</mod>
132                 <mod type="const">4,020600e-07</mod>
133             </seqstat>
134         </operation>
135         <operation name="INT_loc_arr ADD" type="complex" dim="1">
136             <single>
137                 <mod type="add_nr" nr="1">5,732000e-08</mod>
138                 <mod type="mul_nr" nr="1">7,800000e-08</mod>
139             </single>
140         </operation>
141         <operation name="INT_glob_var ADD">
142             <single>
143                 <base>2,889270e-08</base>
144                 <mod type="loc_var">2,851290e-08</mod>
145                 <mod type="const">1,651650e-08</mod>
146             </single>
147             <seqop nr="2">
148                 <base>3,489270e-08</base>
149             </seqop>
150             <seqop nr="3">
151                 <base>3,976470e-08</base>
152             </seqop>
153             <seqop nr="5">
154                 <base>4,876560e-08</base>
155             </seqop>
156             <seqop nr="10">
157                 <base>8,145130e-08</base>
158             </seqop>
159             <seqstat nr="2">
160                 <base>4,051350e-08</base>
161                 <mod type="loc_var">4,051530e-08</mod>
162                 <mod type="const">3,901620e-08</mod>
163             </seqstat>
164             <seqstat nr="3">
165                 <base>4,951470e-08</base>
166                 <mod type="loc_var">5,101470e-08</mod>
167                 <mod type="const">4,501500e-08</mod>
168             </seqstat>
169             <seqstat nr="5">
170                 <base>7,202670e-08</base>
171                 <mod type="loc_var">6,751410e-08</mod>
172                 <mod type="const">7,126500e-08</mod>
173             </seqstat>
174             <seqstat nr="10">
```

```
175             <base>1,339930e-07</base>
176             <mod type="loc_var">1,350400e-07</mod>
177             <mod type="const">1,286100e-07</mod>
178         </seqstat>
179     </operation>
180     <operation name="INT_glob_arr ADD" type="simple" dim="1">
181         <single>
182             <base>3,480000e-08</base>
183             <mod type="var">3,480000e-08</mod>
184             <mod type="loc_arr">4,734000e-08</mod>
185             <mod type="loc_var">2,856000e-08</mod>
186             <mod type="const">2,634000e-08</mod>
187         </single>
188         <seqop nr="2">
189             <base>4,425000e-08</base>
190         </seqop>
191         <seqop nr="3">
192             <base>5,106000e-08</base>
193         </seqop>
194         <seqstat nr="2">
195             <base>5,556000e-08</base>
196             <mod type="loc_var"> 6,828000e-08</mod>
197             <mod type="const">4,203000e-08</mod>
198         </seqstat>
199         <seqstat nr="3">
200             <base>7,986000e-08</base>
201             <mod type="loc_var">6,009000e-08</mod>
202             <mod type="const">5,817000e-08</mod>
203         </seqstat>
204         <seqstat nr="5">
205             <base>1,251600e-07</base>
206             <mod type="loc_var">9,576000e-08</mod>
207             <mod type="const">9,159000e-08</mod>
208         </seqstat>
209     </operation>
210         <single>
211             <base>6,951900e-08</base>
212             <mod type="var">6,078900e-08</mod>
213             <mod type="const">5,050500e-08</mod>
214         </single>
215         <seqop nr="2">
216             <base>1,356675e-07</base>
217         </seqop>
218         <seqop nr="3">
219             <base>1,079920e-07</base>
220         </seqop>
221         <seqstat nr="2">
```

```
222             <base>1,369524e-07</base>
223             <mod type="var">1,197543e-07</mod>
224             <mod type="const">9,949490e-08</mod>
225         </seqstat>
226         <seqstat nr="3">
227             <base>2,016800e-07</base>
228             <mod type="var">1,756659e-07</mod>
229             <mod type="const">1,456200e-07</mod>
230         </seqstat>
231         <seqstat nr="5">
232             <base>3,286800e-07</base>
233             <mod type="var">2,584320e-07</mod>
234             <mod type="const">2,068500e-07</mod>
235         </seqstat>
236         <seqstat nr="10">
237             <base>6,321540e-07</base>
238             <mod type="var">5,059440e-07</mod>
239             <mod type="const">4,020600e-07</mod>
240         </seqstat>
241     </operation>
242     <operation name="INT_glob_arr ADD" type="complex" dim="1">
243         <single>
244             <mod type="add_nr" nr="1">5,058000e-08</mod>
245             <mod type="mul_nr" nr="1">5,178000e-08</mod>
246         </single>
247     </operation>
248     <operation name="INT_par_var ADD">
249         <single>
250             <base>1,980900e-08</base>
251             <mod type="loc_var">1,365300e-08</mod>
252             <mod type="const">1,365300e-08</mod>
253         </single>
254         <seqop nr="2">
255             <base>2,391080e-08</base>
256         </seqop>
257         <seqop nr="3">
258             <base>2,719930e-08</base>
259         </seqop>
260         <seqop nr="5">
261             <base>3,340650e-08</base>
262         </seqop>
263         <seqstat nr="2">
264             <base>3,3309900e-08</base>
265             <mod type="loc_var">2,296550e-08</mod>
266             <mod type="const">2,284550e-08</mod>
267         </seqstat>
268         <seqstat nr="3">
```

```
269             <base>4,838970e-08</base>
270             <mod type="loc_var">3,339700e-08</mod>
271             <mod type="const">3,328650e-08</mod>
272         </seqstat>
273         <seqstat nr="5">
274             <base>7,876200-08</base>
275             <mod type="loc_var">5,431410e-08</mod>
276             <mod type="const">5,436500e-08</mod>
277         </seqstat>
278     </operation>
279     <operation name="INT_par_arr ADD" type="simple" dim="1">
280         <single>
281             <base>3,618000e-08</base>
282             <mod type="var">3,474000e-08</mod>
283             <mod type="loc_arr">3,375000e-08</mod>
284             <mod type="loc_var">2,571000e-08</mod>
285             <mod type="const">2,493000e-08</mod>
286         </single>
287         <seqop nr="2">
288             <base>4,680000e-08</base>
289         </seqop>
290         <seqop nr="3">
291             <base>6,063000e-08</base>
292         </seqop>
293         <seqstat nr="2">
294             <base>6,093000e-08</base>
295             <mod type="loc_var"> 4,374000e-08</mod>
296             <mod type="const">4,140000e-08</mod>
297         </seqstat>
298         <seqstat nr="3">
299             <base>8,847000e-08</base>
300             <mod type="loc_var">6,288000e-08</mod>
301             <mod type="const">5,838000e-08</mod>
302         </seqstat>
303         <seqstat nr="5">
304             <base>1,442100e-07</base>
305             <mod type="loc_var">9,843000e-08</mod>
306             <mod type="const">9,060000e-08</mod>
307         </seqstat>
308         <seqstat nr="10">
309             <base>2,745300e-07</base>
310             <mod type="loc_var">1,880136e-07</mod>
311             <mod type="const">1,726415e-07</mod>
312         </seqstat>
313     </operation>
314     <operation name="INT_loc_arr ADD" type="simple" dim="3">
315         <single>
```

```
316             <base>6,075900e-08</base>
317             <mod type="loc_var">4,971300e-08</mod>
318             <mod type="const">4,364400e-08</mod>
319         </single>
320         <seqop nr="2">
321             <base>1,356675e-07</base>
322         </seqop>
323         <seqop nr="3">
324             <base>1,079920e-07</base>
325         </seqop>
326         <seqstat nr="5">
327             <base>2,724120e-07</base>
328             <mod type="loc_var">2,350980e-07</mod>
329             <mod type="const">1,829460e-07</mod>
330         </seqstat>
331         <seqstat nr="10">
332             <base>5,435100e-07</base>
333             <mod type="loc_var">4,110840e-07</mod>
334             <mod type="const">3,586260e-07</mod>
335         </seqstat>
336     </operation>
337     <operation name="INT_par_arr ADD" type="complex" dim="1">
338         <single>
339             <mod type="add_nr" nr="1">4,974000e-08</mod>
340             <mod type="mul_nr" nr="1">6,318000e-08</mod>
341         </single>
342     </operation>
343 </platform>
```

Dodatak B

Profil aplikacije AES_G

```
1  <?xml version="1.0"?>
2  <application  name="AES_G">
3      <procedure name="KeyExpansion">
4          <loop count="4">
5              <operation class="MEM_glob_var" type="ASSIGN" mod="const">1</
                   operation>
6              <operation class="MEM_glob_arr" type="ASSIGN" index_type="complex
                   " dim="1" mul_nr="1">1</operation>
7              <operation class="MEM_glob_arr" type="ASSIGN" index_type="complex
                   " dim="1" mul_nr="1" add_nr="1">3</operation>
8          </loop>
9          <loop count="40">
10             <loop count="4">
11                 <operation class="MEM_glob_var" type="ASSIGN" mod="const">1</
                       operation>
12                 <operation class="MEM_glob_arr" type="ASSIGN" index_type="
                       complex" dim="1" mul_nr="1" add_nr="2">3</operation>
13             </loop>
14             <branch cond="i%4==0">
15                 <truebody>
16                     <operation class="MEM_glob_var" type="ASSIGN" mod="const"
                           >1</operation>
17                     <operation class="MEM_glob_arr" type="ASSIGN" mod="
                           loc_var" index_type="const" dim="1">1</operation>
18                     <operation class="MEM_glob_arr" type="ASSIGN" index_type=
                           "const" dim="1">3</operation>
19                     <operation class="MEM_glob_arr" type="ASSIGN" mod="
                           loc_var" index_type="const" dim="1">1</operation>
20                     <operation class="MEM_glob_arr" type="ASSIGN" index_type=
                           "const" dim="1">4</operation>
21                     <operation class="MEM_glob_arr" type="PROC">4</operation>
22                 </truebody>
23                 <falsebody>
```

```
24                      <branch cond="4&gt;6 &amp;&amp; i%4==4">
25                          <truebody>
26                              <operation class="MEM_glob_var" type="ASSIGN" mod
                                    ="const">1</operation>
27                              <operation class="MEM_glob_arr" type="ASSIGN"
                                    index_type="const" dim="1">4</operation>
28                              <operation class="MEM_glob_arr" type="PROC">4</
                                    operation>
29                          </truebody>
30                          <falsebody>
31                          </falsebody>
32                      </branch>
33                  </falsebody>
34              </branch>
35              <operation class="MEM_glob_var" type="ASSIGN" mod="const">1</
                    operation>
36              <operation class="LOG_glob_arr" type="LOG" index_type="complex"
                    dim="1" mul_nr="1" add_nr="1">4</operation>
37              <operation class="INT_glob_var" type="ADD">1</operation>
38          </loop>
39      </procedure>
40      <procedure name="AddRoundKey">
41          <loop count="4">
42              <operation class="MEM_glob_var" type="ASSIGN" mod="const">1</
                    operation>
43              <loop count="4">
44                  <operation class="MEM_glob_var" type="ASSIGN" mod="const">1</
                        operation>
45                  <operation class="LOG_glob_arr" type="LOG" index_type="
                        complex" dim="1" mul_nr="1" add_nr="1">4</operation>
46              </loop>
47          </loop>
48      </procedure>
49      <procedure name="SubBytes">
50          <loop count="4">
51              <operation class="MEM_glob_var" type="ASSIGN" mod="const">1</
                    operation>
52              <loop count="4">
53                  <operation class="MEM_glob_var" type="ASSIGN" mod="const">1</
                        operation>
54                  <operation class="MEM_glob_arr" type="ASSIGN" index_type="
                        complex" dim="1" mul_nr="1" add_nr="1">1</operation>
55                  <operation class="MEM_glob_arr" type="PROC">1</operation>
56              </loop>
57          </loop>
58      </procedure>
59      <procedure name="ShiftRows">
```

```
60          <operation class="MEM_glob_arr" type="ASSIGN" mod="loc_var"
                index_type="const" dim="1">1</operation>
61          <operation class="MEM_glob_arr" type="ASSIGN" index_type="const" dim=
                "1">3</operation>
62          <operation class="MEM_glob_arr" type="ASSIGN" mod="loc_var"
                index_type="const" dim="1">2</operation>
63          <operation class="MEM_glob_arr" type="ASSIGN" index_type="const" dim=
                "1">1</operation>
64          <operation class="MEM_glob_arr" type="ASSIGN" mod="loc_var"
                index_type="const" dim="1">2</operation>
65          <operation class="MEM_glob_arr" type="ASSIGN" index_type="const" dim=
                "1">1</operation>
66          <operation class="MEM_glob_arr" type="ASSIGN" mod="loc_var"
                index_type="const" dim="1">2</operation>
67          <operation class="MEM_glob_arr" type="ASSIGN" index_type="const" dim=
                "1">3</operation>
68          <operation class="MEM_glob_arr" type="ASSIGN" mod="loc_var"
                index_type="const" dim="1">1</operation>
69      </procedure>
70      <procedure name="MixColumns">
71          <loop count="4">
72              <operation class="MEM_glob_var" type="ASSIGN" mod="const">1</
                    operation>
73              <operation class="MEM_glob_arr" type="ASSIGN" mod="loc_var"
                    index_type="simple" dim="1">1</operation>
74              <operation class="LOG_glob_arr" type="LOG" mod="loc_var"
                    index_type="simple" dim="1">3</operation>
75              <operation class="LOG_glob_arr" type="LOG" mod="loc_var"
                    index_type="simple" dim="1" >1</operation>
76              <operation class="MEM_loc_var" type="ASSIGN">1</operation>
77              <operation class="MEM_loc_var" type="PROC">1</operation>
78              <operation class="LOG_glob_arr" type="LOG" mod="loc_var"
                    index_type="simple" dim="1">2</operation>
79              <operation class="LOG_glob_arr" type="LOG" mod="loc_var"
                    index_type="simple" dim="1" >1</operation>
80              <operation class="MEM_loc_var" type="ASSIGN">1</operation>
81              <operation class="MEM_loc_var" type="PROC">1</operation>
82              <operation class="LOG_glob_arr" type="LOG" mod="loc_var"
                    index_type="simple" dim="1">2</operation>
83              <operation class="LOG_glob_arr" type="LOG" mod="loc_var"
                    index_type="simple" dim="1" >1</operation>
84              <operation class="MEM_loc_var" type="ASSIGN">1</operation>
85              <operation class="MEM_loc_var" type="PROC">1</operation>
86              <operation class="LOG_glob_arr" type="LOG" mod="loc_var"
                    index_type="simple" dim="1">2</operation>
87              <operation class="LOG_glob_arr" type="LOG" mod="loc_var"
                    index_type="simple" dim="1" >1</operation>
```

```
88              <operation class="MEM_loc_var" type="ASSIGN">1</operation>
89              <operation class="MEM_loc_var" type="PROC">1</operation>
90              <operation class="LOG_glob_arr" type="LOG" mod="loc_var"
                    index_type="simple" dim="1">2</operation>
91          </loop>
92      </procedure>
93      <procedure name="Cipher">
94          <loop count="4">
95              <operation class="MEM_glob_var" type="ASSIGN" mod="const">1</
                    operation>
96              <loop count="4">
97                  <operation class="MEM_glob_var" type="ASSIGN" mod="const">1</
                        operation>
98                  <operation class="MEM_glob_arr" type="ASSIGN" index_type="
                        complex" dim="1" mul_nr="1" add_nr="1">1</operation>
99              </loop>
100         </loop>
101         <procedure name="AddRoundKey"/>
102         <loop count="9">
103             <procedure name="SubBytes"/>
104             <procedure name="ShiftRows"/>
105             <procedure name="MixColumns"/>
106             <procedure name="AddRoundKey"/>
107         </loop>
108         <procedure name="SubBytes"/>
109         <procedure name="ShiftRows"/>
110         <procedure name="AddRoundKey"/>
111         <loop count="4">
112             <operation class="MEM_glob_var" type="ASSIGN" mod="const">1</
                    operation>
113             <loop count="4">
114                 <operation class="MEM_glob_var" type="ASSIGN" mod="const">1</
                        operation>
115                 <operation class="MEM_glob_arr" type="ASSIGN" index_type="
                        complex" dim="1" mul_nr="1" add_nr="1">1</operation>
116             </loop>
117         </loop>
118     </procedure>
119     <procedure name="main">
120         <procedure name="KeyExpansion"/>
121         <loop count="2">
122             <loop count="16">
123                 <operation class="LOG_glob_arr" type="LOG" mod="loc_arr"
                        index_type="simple" dim="1">1</operation>
124             </loop>
125             <procedure name="Cipher"/>
126             <loop count="16">
```

```
127                 <operation class="MEM_glob_arr" type="ASSIGN"  mod="loc_arr"
                        index_type="simple" dim="1">1</operation>
128             </loop>
129         </loop>
130     </procedure>
131 </application>
```

Životopis

Nikolina Frid rođena je 1988. godine u Zagrebu. Sveučilišni diplomski studij računarstva završila je 2013. godine na Fakultetu elektrotehnike i računarstva Sveučilišta u Zagrebu s najvišom pohvalom (*summa cum laude*). Dobitnica je nekoliko stipendija i nagrada za istaknuti uspjeh tijekom studija među kojima su Stipendija Grada Zagreba te Brončana plaketa Josip Lončar. Od 2013. godine zaposlena je na Sveučilištu u Zagrebu, Fakultetu elektrotehnike i računarstva. Do 2016. godine radila je kao suradnik na međunarodnom FP7 projektu pod nazivom „*Embedded Computer Engineering Learning Platform (E2LP)*“, a od 2016. do danas radi u suradničkom zvanju asistent na Zavodu za elektroniku, mikroelektroniku, računalne i inteligentne sustave istog Fakulteta. Znanstveni i stručni interesi uključuju arhitekturu računalnih sustava s naglaskom na heterogene višeprocesorske sustave, evolucijsko računarstvo i algoritme strojnog učenja. Na temelju znanstvenog rada tijekom doktorskog studija do sada je objavila desetak radova u zbornicima međunarodnih znanstvenih skupova i časopisima. Govori engleski i francuski jezik.

Popis objavljenih radova

1. Frid, Nikolina; Ivošević, Danko; Sruk, Vlado. Elementary operations: a novel concept for source-level timing estimation. *Automatika*, Vol. 60, Issue 1, (2019). 91-104.
2. Frid, Nikolina; Sruk, Vlado. Memory-aware multiobjective design space exploration of heteregeneous MPSoC. *Proceedings of 41st International Convention on Information and Communication Technology, Electronics and Microelectronics* (2018). 861-866
3. Frid, Nikolina; Ivošević, Danko; Sruk, Vlado. Performance Estimation in Heterogeneous MPSoC Based on Elementary Operation Cost. *Proceedings of 39th International Convention on Information and Communication Technology, Electronics and Microelectronics* (2016). 1409-1412
4. Ivošević, Danko; Frid, Nikolina; Sruk, Vlado. Function-level Performance Estimation for

Heterogeneous MPSoC Platforms. *Proceedings of ZINC 2016* (2016). 37-40

5. Frid, Nikolina; Ivošević, Danko; Sruk, Vlado. Heterogeneity Impact on MPSoC Platforms Performance. *Proceedings of MIPRO 2015 38th International Convention* (2015). 1274-1279
6. Žagar, Martin; Frid, Nikolina; Knezović, Josip; Hofman, Daniel; Kovač, Mario; Sruk, Vlado; Mlinarić, Hrvoje. Work in Progress: Embedded Computer Engineering Learning Platform Capabilities. *Proceedings of 2015 IEEE Global Engineering Education Conference* (2015). 737-739
7. Ivošević, Danko; Frid, Nikolina. Performance-Occupation Trade-off Examination in Custom Processor Design. *Proceedings of MIPRO 2014 37th International Convention* (2014). 1258-1263
8. Frid, Nikolina; Sruk, Vlado. Critical Path Method Based Heuristics for Mapping Application Software onto Heterogeneous MPSoCs. *Proceedings of MIPRO 2014 37th International Convention* (2014). 1264-1268
9. Frid, Nikolina; Sruk, Vlado; Mlinarić, Hrvoje; Kovač, Mario. Computer Engineering Laboratory Course: E2LP Platform Experience. *Proceedings of the E2LP 2014 Workshop* (2014). 5-8
10. Žagar, Martin; Frid, Nikolina; Knezović Josip; Hofman, Daniel; Kovač, Mario; Sruk, Vlado; Mlinarić Hrvoje. Unified, Multiple Target, Computer Engineering Learning Platform - Design Results and Learning Outcomes. *Proceedings IEEE Global Engineering Education Conference EDUCON 2014* (2014). 926-929
11. Frid, Nikolina; Sruk, Vlado; Mlinarić, Hrvoje. Open Source SaaS Cloud Platforms: Concepts, Virtualization Overhead and Deployment Issues. *Proceedings of 24th Central European Conference on Information and Intelligent Systems CECIIS 2013* (2013). 106-112
12. Frid, Nikolina; Mlinarić, Hrvoje; Knezović, Josip. Acceleration of DCT transformation in JPEG image conversion. Proceedings of MIPRO 2013 36th International Convention (2013). 1693-1696

Biography

Nikolina Frid was born in 1988 in Zagreb. She graduated in computer science in 2013 at the Faculty of Electrical Engineering and Computing, University of Zagreb, with the highest accolades (summa cum laude). She has received several scholarships and awards for distinguished success during her studies, including the Zagreb City Scholarship and the Josip Lončar Bronze Plaque. Since 2013, she has been employed at the University of Zagreb, Faculty of Electrical Engineering and Computing. Until 2016, she worked as an associate at the international FP7 project, and from 2016 until today she has been working as a teaching and research assistant at the Department of Electronics, Microelectronics, Computer and Intelligent Systems of the same Faculty. Her scientific and professional interests are focused on computer systems architecture with an emphasis on heterogeneous multiprocessor systems, evolutionary computing, and machine learning algorithms. Based on her scientific work during her doctoral studies, she has so far published a dozen papers in proceedings of international scientific conferences and journals. She speaks English and French.