University of Zagreb

FACULTY OF ORGANIZATION AND INFORMATICS

Ivan Švogor

A FRAMEWORK FOR ALLOCATION OF SOFTWARE COMPONENTS ONTO A HETEROGENEOUS COMPUTING SYSTEM

DOCTORAL THESIS

Varaždin, 2016

Sveučilište u Zagrebu

FAKULTET ORGANIZACIJE I INFORMATIKE

Ivan Švogor

OKVIR ZA ALOKACIJU SOFTVERSKIH KOMPONENATA NA HETEROGENOJ RAČUNALNOJ PLATFORMI

DOKTORSKI RAD

Varaždin, 2016.

University of Zagreb

FACULTY OF ORGANIZATION AND INFORMATICS

Ivan Švogor

A FRAMEWORK FOR ALLOCATION OF SOFTWARE COMPONENTS ONTO A HETEROGENEOUS COMPUTING SYSTEM

DOCTORAL THESIS

Supervisors:
professor Neven Vrček
professor Ivica Crnković

Varaždin, 2016

“Tis but a scratch...”
Monty Python

Acknowledgments

In the last few years, during the “PhD journey”, I’ve worked with interesting people from all over the world, and since this is the Acknowledgment section it is customary to thank most memorable people along that journey. I’m keeping it as short as possible.

This journey involved a one year stay at Mälardalen University in Sweden which, except learning a great deal, turned out to be a great character building experience. The details and reasons I’ll keep for myself at this point, however, people who know me best surely agree. Since this was possible only thanks to my supervisors, Ivica and Neven, I’m taking this opportunity to thank both of them, for the many hours of talking and working, lessons and patience you had with me. Thanks!

Furthermore, both in Croatia and in Sweden I got to work with good colleagues, professors and fellow PhD students with whom I spent many hours of talking, sharing delights and frustrations of being a PhD student, coding, soldering, writing research papers and a lot, a lot of coffee drinking. Thanks to everyone of you who is reading this, especially to my office mates, team mates and friends.

This thesis would never take place without the support and patience from my parents and my sister. Thanks guys! At last, thank you Antonija, for sharing this journey with me, following me at the crossroads and smoothing the bumps along the road.

Varaždin, January 2016

ABSTRACT

A recent development of heterogeneous platforms (i.e. those containing different types of computing units such as multicore CPUs, GPUs, and FPGAs) has enabled significant improvements in performance of real-time data processing. However, due to increased development efforts for such platforms, they are not fully exploited. To use the full potential of such platforms, we need new frameworks and methods for capturing the optimal configuration of the software. Different configurations, i.e. allocations of software components to different computing unit types can be essential for getting the maximal utilization of the platform. For more complex systems it is difficult to find ad–hoc, good enough or the best configuration.

This research suggests the application of component based software engineering (CBSE) principles, by which it is possible to achieve the same functionality of software components across various computing units of different types, however with different extra–functional properties (EFP). The objective of this research is to construct a framework which optimizes the allocation of software components on a heterogeneous computing platform with respect to specified extra-functional requirements.

The I-IV allocation framework, proposed by this research, consist of formalisms necessary for modeling of a heterogeneous computing platform and exploring the design space, which results with an optimal design decision. The I-IV allocation framework was verified in two steps, focusing on two EFPs; the average power consumption and the average execution time. The experimental platform was a tracked robot, developed for the purpose of this research. It contains a CPU, a GPU and an FPGA, along with 32 software components deployable onto these units. Both steps resulted in a positive result confirming the claim that the I-IV framework, along with its Component allocation model $\mathbb{M}_\alpha$ correctly represents the heterogeneous system performance, with consideration to multiple criteria.

Prošireni sažetak

Usprkos tome da je u posljednjih nekoliko godina povećanje radnog takta središnje procesne jedinice (CPU) usporeno, ako ne i zaustavljeno, performanse suvremenih računala i dalje rastu, ali ne zbog radnog takta. To znači da se i performanse računalnih programa više na ovaj način ne mogu unaprijediti, čak što više, daljnje povećavanje radnog takta CPU-a pokazalo se neučinkovitim. Zbog toga, došlo je do suštinske promjene u građi procesora, odnosno to repliciranja procesnih jezgri te ugradbom dodatnih namjenskih procesnih jedinica koje su specijalizirane za određeni tip zadataka. Najčešće su to grafička procesna jedinica (GPU), programirljiva polja logičkih blokova (FPGA), integrirani krugovi specifične namjene (ASIC), itd. Istovremeno, zajednica prepoznala je veliki istraživački potencijal heterogenih računalnih sustava, odnosno sustava sa mnoštvom procesnih jedinica različitog tipa, obzirom da omogućuju izuzetna poboljšanja performansi softvera.

Mnogi se istraživači već dulje vrijeme bave heterogenim računalstvom, što znači da to nije nova ideja, no u posljednjih nekoliko godina, zbog fizičkih ograničenja vezanih uz arhitekturu procesnih jedinica, heterogeno računalstvo postaje sve popularnija istraživačka tema. Uz izuzetno povećanje procesne moći, heterogeno računalstvo donosi i mnogo izazova, prvenstveno za softverske inženjere. Naime, razvoj softvera za takve sustave vrlo je zahtjevan zbog primjerice, potrebe za rukovanjem sa više različitih tipova podataka ili programskih jezika unutar istog računalnog programa, kompatibilnosti pojedinih procesnih jedinica i konverzije tipova podataka, potrebe za specijaliziranim bibliotekama koda, korištenja različitih struktura podataka kroz više arhitekturalni slojeva računala i računalnog programa, itd. Osim toga, obzirom na to da se heterogeni sustavi prvenstveno koriste kao elementi ugradbenih računala u industriji, softverski inženjeri uz funkcionalne zahtjeve, dodatnu pozornost moraju dati ne-funkcionalnim zahtjevima (EFP).

Kako bi se upravljalo funkcionalnim i ne-funkcionalnim zahtjevima softvera, u složenim heterogenim računalnim sustavima, često se primjenjuju načela komponentno orijentiranog softverskog inženjerstva (CBSE), koja su u softverskoj zajednici dobro poznata i dokazana. CBSE obuhvaća modele, metode i smjernice za softverske inženjere koji razvijaju sustave temeljene na komponentama, odnosno građevnim jedinicama koje komuniciraju putem ugovorno definiranih sučelja, koje se mogu samostalno ugrađivati i jednostavno zamjenjivati. Time, CBSE daje snažne temelje za prethodno spomenute vezane uz razvoj softvera namijenjenog za heterogene računalne sustave. U tom kontekstu, CBSE omogućuje postizanje jednake funkcionalnosti komponenata

softvera alociranih na (različite) procesne jedinice (različitog tipa), no sa drugačijim ne-funkcionalnim svojstvima. To znači da pojedine alokacije komponenata softvera mogu biti više ili manje učinkovite obzirom na scenarije njihove primjene, odnosno njihove ulazne parametre, što za sobom povlači i pitanje ukupnih performansi sustava. Prema tome, zadatak arhitekta softvera najprije je definirati svojstva najbolje alokacije obzirom na više kriterija, poput dostupnosti resurs, ne-funkcionalna svojstva i ograničenja, a potom na konkretnoj heterogenoj računalnoj platformi učinkovito i pronaći takvu alokaciju.

Temeljni cilj ovog istraživanja je konstruirati okvir za optimizaciju alokacije komponenti softvera na heterogenoj računalnoj platformi, koji uzimajući u obzir ograničenja resursa dostupnih na računalnim jedinicama (različitog tipa), specifikacije komponenata softvera i ograničenja koja definira arhitekt sustava učinkovito pronalazi najbolju alokaciju. Ova disertacija predlaže Alokacijski okvir I-IV sastavljen od formalnih elemenata koji omogućuju stvaranje modela heterogenog računalnog sustava te pretraživanje prostora potencijalnih alokacija, te definira korake kojima se postiže optimalna arhitektura sustava. Kako u ovom slučaju prostor potencijalnih rješenja, odnosno alokacija eksponencijalno raste (uz m dostupnih računalnih jedinica te n dostupnih komponenti softvera, prostor rješenja je m^n), razvijen je i prototip alata koji automatizira Alokacijski okvir I-IV, što je inače dugotrajan ili čak neizvediv proces. Za opis ne–funkcionalnih svojstava heterogenih sustava, koristi se Model za alokaciju komponenata $\mathbb{M}_\alpha$. Taj model, primjenom težinske funkcije w omogućuje kvantifikaciju pojedinih alokacija čime je omogućena njihova usporedba te procjena prikladnosti korištenja istih. Istovremeno, težinska funkcija w daje uvid u performanse sustava u njegovoj ranoj fazi razvoja (čak prije nego su komponente razvijene).

Vjerodostojnost Alokacijskog okvira I-IV provjerena je u dva koraka (eksperimenta), pri čemu je fokus bio na dva ne-funkcionalna svojstva sustava: prosječni električni učinak električne struje i prosječno vrijeme izvođenja operacija softvera. Eksperimentalna platforma bila su robotska kolica sa heterogenim računalnim sustavom sačinjenim od CPU-a, GPU-a te FPGA-a, zajedno sa tridesetak komponenata softvera koje je moguće alocirati na te računalne jedinice.

Prvi korak provjere odnosio se na provjeru točnosti, odnosno procjenu prikladnosti težinske funkcije w da kvantificira performanse pojedine alokacije. Postupak je proveden primjenom šest različitih alokacija koje predstavljaju dva različita scenarija izvođenja. Odabrane alokacije, nakon što su kvantificirane težinskom funkcijom w, zapisane su tablično i rangirane prema predviđenim performansama. Nakon toga, te iste alokacije su implementirane na stvarnom sustavu, ranije spomenutim robotskim kolicima. Iscrpnim mjerenje (u intervalu pouzdanosti od 95%), zabilježene su performanse alokacija i ponovno su rangirane u rang listu. Rezultat oba rangiranja bio je jednak, čime slijedi da model za raspodjelu komponenata $\mathbb{M}_\alpha$, te njegova težinska funkcija w mogu korektno predvidjeti performanse pojedine alokacije u realnom sustavu. Ovakav ishod, doveo

je do drugog koraka provjere koji se odnosi na scenarij(e) u kojem postoji izrazito veliki broj komponenti softvera te računalnih jedinica, čime prostor potencijalnih rješenja postaje toliko velik pronalaženje najbolje alokacije metodom iscrpnog pretraživanja nije moguće učinkovito provesti.

Obzirom da trenutna implementacija Alokacijskog modela I-IV definira heurističke metode za rješavanje navedenog problema, drugi korak provjere za cilj ima procijeniti sub-optimalno rješenje genetskog algoritma i metode simuliranog kaljenja. Uz heurističke metode, generirane su i proizvoljne alokacije, jer u nekim slučajevima su takve alokacije podjednako dobre ili čak bolje od heurističkih metoda. U prvoj iteraciji, provjeravala se preciznost navedenih metoda, odnosno njihovo odstupanje od optimalne alokacije dane iscrpnim pretraživanje u prostoru do 5^{12}. Pokazalo se kako genetski algoritam daje najbolja rješenja, odnosno alokacije koje minimalno odstupaju od optimalnog rješenja. Nadalje, za prostore rješenja između 10^{20} do 30^{70} gdje iscrpno pretraživanje nije učinkovito, usporedba je pokazala da obje heurističke metode daju bolja sub-optimalne alokacije od proizvoljno definiranih alokacija i to u najkraćem vremenu. Iako je statistički vjerojatno, ni u jednom slučaju (u 55 ponavljanja, s povećavanjem prostora rješenja) nije zabilježeno da proizvoljno generirana alokacija daje bolje performanse od alokacije dobivene predloženim heurističkim metodama, čime je završila validacija predloženog okvira i svih njegovih elemenata.

Keywords

heterogeneous computing platform, optimal software allocation, cyber–physical system, formal model, software performance measurement

GLOSSARY

heterogeneous platform	a computing system which contains multiple processing units of different types (it can also be referred to as heterogeneous computing platform or heterogeneous system architecture (HSA))
heterogeneous computing	a research discipline which deals with heterogeneous platforms
CPU	central processing unit
GPU	graphical processing unit
FPGA	field programmable gate array
ASIC	application specific integrated circuit
CBSE	component based software engineering
EFP	extra–functional property, a specification of a system which describes how should a system perform certain actions.
UML	unified modeling language.
pipe–line style	software architecture style in which the output of a previous process is the input for the next process.
confidence interval	a measure for expressing the uncertainty associated with a sample estimate of a population mean.
SWEBOK	software engineering body of knowledge.
OpenCL	open computing language library
OpenCV	open computing vision library
CLB	configurable logic block
allocation	an allocation is a mapping of all n software components from the set $\mathcal{C}$ to a (sub–) set of computing units $\mathcal{U}'$
I-IV framework	component allocation framework

TiWo	a tracked robot, real–world platform containing a CPU, a GPU and an FPGA.
S,G,E,D,H	a combination of image filters executed in the given order, each letter signifying the name of the filter. In this particular case, Sobel, Gauss, Erode, Dilate, Hystogram.
GA	a method for obtaining the optimal software allocation which uses an exhaustive search.
GA	a method for obtaining the (sub–) optimal software allocation which uses a genetic algorithm.
SA	a method for obtaining the (sub–) optimal software allocation which uses simulated annealing
RAND	randomly generated software allocation

Contents

List of Figures

List of Tables

CHAPTER 1

INTRODUCTION

This chapter introduces the motivation and the research objectives of this thesis. Based on these objectives, three research questions are derived which represent the fundamental challenges of this work. Moreover, two hypothesis are made to suggest the possible solutions to these challenges and also guide the research process. The following sections emphasize the contribution of this research, its scope along with the research method. The chapter ends by outlining the organization of the thesis.

During the 90s and early 2000s speed–ups of software largely relied on increasing of central processing units (CPU) clock rate. When this became inefficient in mid 2000s due to technological issues related to CPU chip design (heat dissipation, leakage currents, memory handling, etc.) a new paradigm of chip design was inevitable. It resulted replication of processing cores within a single chip which enabled true parallel program execution. The further development of this technology and the present day high demand for data processing led to new considerations for performance enhancement which eventually resulted in *heterogeneous computing*[1, 43].

Heterogeneous computing refers to systems which utilize more than one type of processing units. Each computing unit type has different properties and is specialized for specific types of tasks. Most common heterogeneous computing platforms consist of CPUs, GPUs and FPGAs. Graphical processing unit (GPU) is a high performance system capable of massive parallelism which CPUs traditionally cannot handle. However, its processing benefits come at a higher power consumption price. To overcome this issue, computer engineers often resort to use field programmable gate arrays (FPGAs) which are essentially *empty* chips programmable for any purpose. Historically, FPGAs were largely used for highly specific tasks by aerospace industry, today they are the large part of embedded systems domain. Despite the previously mentioned advantages, compared to CPUs and GPUs their appeal lags in unit cost and demanding development efforts for even basic functionality.

In spite of advantages of heterogeneous platforms, they also carry new challenges for software engineers. Software development for heterogeneous platforms is a complex endeavor and the developers must handle communication and synchronization of different computing unit types, convert data between different data types and structures,

use multiple programming languages, use specialized libraries etc. Additionally, this is accompanied by traditional software engineering objectives like fulfilling the specified functional requirements and conforming to extra–functional requirements.

1.1 Objective

In order to address issues described in the previous section and to handle such software complexity this research proposes the application of *component based software engineering* principles (CBSE) [27]. The main concern of CBSE is to deal with dynamic configurations, variant explosions, scalability and reusability. CBSE provides methods, models and guidelines for developers of component based systems. Building blocks of such systems are software components, which have the following properties: a) contractually specified interfaces, b) independent deployability and c) replaceability. As such CBSE provides a solid foundation to address challenges of heterogeneous computing.

The result of applying CBSE principles to software for heterogeneous computing is a system composed of manageable building blocks, which are independent, replaceable, reusable and enable various configurations based on the specification of extra–functional properties (EFPs)[1]; e.g. energy consumption, processing time, memory requirements, reliability, safety, security, maintainability, etc. By applying CBSE principles to heterogeneous computing, it is possible to achieve the same functionality of software components across various computing units of different types, however with different EFPs. Consequently, some configurations can result in poor performance of software, while others can result in great performance despite using advanced hardware [100]. Hence, the *overall objective* of this research is to:

Construct a framework which optimizes the allocation of software components on a heterogeneous computing platform with respect to specified extra-functional requirements.

The specific research goals derived from overall objective are the following:

RG-1: *Develop a model which describes software components and a heterogeneous computing platform.*

Despite the existence of many component models, most of them do not address heterogeneous computing platforms. The model developed in this research needs to enable the formal specification of a software architecture and its requirements along with the formal specification of a hardware platform with the resources it provides.

RG-2: *Define a framework for comparison of software allocations on a specific heterogeneous platform.*

[1] Extra–functional properties – also referred to as Quality of Service, non–functional properties or/and attributes [26]. Although there are subtle differences between these terms [25], in this work they will be treated as synonyms.

In order to find the optimal configuration, i.e. allocation of software components on a targeted computing platform, one needs a framework which evaluates particular allocations. By comparing feasible allocation alternatives, the goal is to find the one which conforms to the requirements of the software system considering all multiple criteria like energy efficiency, memory efficiency, processing speed, communication bandwidth etc., while providing the best overall system performance.

RG-3: *Implement a tool for automatized component allocation based on previous model.*

With a large number of computing units and software components the design space of a system grows exponentially so manual methods for finding an optimal allocation are not efficient. Therefore, this research should deliver a prototype tool which automates this process and provides an insight into the performance of a heterogeneous computing platform in its early development phase; even before the components are developed. This would greatly improve and simplify the allocation decision making for software architect.

1.2 Research questions

This section uses the research challenges introduced in previous sections to elicit concrete research questions, which will provide a foundation and a guideline for further investigation of the presented subject.

The importance of the research in this domain is also recognized by European Commission which is evident by the *Horizon 2020 Work Programme for 2014–15*, i.e. the call for *Advanced Computing* which emphasizes heterogeneous computing as one of the key challenges. Including the integration of hardware and software components and working prototypes with emphasis on low-power, low-cost, security, reliability, scalability etc. [34].

The current approach to software modeling is to apply concepts available in standardized notations, e.g. the most widely known UML[2] (including its extensions and derivatives). This enables software component designers to create compatible components which conform to the same component model. Such models define the form, the interface and the interactions of components. Similarly to this, but from the standpoint of computing hardware, system architects model computing resources and their communication using another family of standardized notations, i.e. architecture description languages (ADLs). To design an entire system, both hardware and software in an early design phase, many modeling approaches can be use (e.g. UML Marte profile, SysML, AADL, etc), however the goal here is to model both hardware and software with considerations to the heterogeneous computing platform while capturing systems' extra–functional properties necessary for architectural decision making and utilize this

[2] http://www.omg.org/spec/UML/

information to suggest best design options. The first research question is as follows:

RQ-1: *How to describe software components and a heterogeneous computing platform using the same model?*

RQ-1.1; *In particular, how can this model represent communication parameters separately from computing parameters?*

RQ-1.2; *How can this model incorporate specific constraints from a software architect?*

RQ-1 entails two sub questions related to communication performance parameters and design constraints imposed by a software architect. If the software allocation decision is made solely based on the computing performance of the computing units, without regard to communication it is likely that such allocation of software components will not result in the best overall performance. It has been shown that the communication bottlenecks are a common issue[54] in CPU–GPU systems, so in order to predict those in an early system design phase, it is necessary for a model to be able to separate communication parameters from computing parameters, hence the *RQ-1.1*. The question *RQ-1.2* emerges from a necessity of a software architect to manually tweak allocation parameters, deliberately prevents allocation of certain components to certain computing units or to deliberately allocate several components together regardless of the computing units.

Since each component allocation largely influences the overall performance of a system, an allocation framework should be developed which would evaluate all the allocation choices and select the best one. Moreover, a software architect using the framework should also be able to specify the meaning of *the best allocation*, i.e. a choice is given to optimize the performance of a designed system with respect to certain EFPs. Consequently, a method for quantifying constraints and extra–functional parameters of an allocation independently of their measurement units is necessary to compare different extra–functional properties and put different allocations into perspective. However, there is an issue with this since different system requirements are measured in different units (e.g. megabytes for memory, watts for energy, time for processing speed). This resulted with a second research question:

RQ-2: *How to measure both the execution performance and communication intensity*[3] *of a particular software allocation?*

Additionally to the previous research question, there should be at least one allocation within the design space of all feasible allocations, which gives the best result with respect to system requirements and constraints. Finding this allocation is hard due to large number of constraints and due to the fact that for with large number of compo-

[3]The communication intensity is defined by the software architect with considerations to a modeled system. It is a discrete numerical value which describes the information exchange between two particular software components, for example a frequency of function calls (a software coupling metric [78]) or an amount of exchanged data.

nents and computing units, design space grows exponentially[4]. This issue brings the final research question:

RQ-3: *How to optimize software component allocation on a heterogeneous platform in the cases with a large design space?*

1.3 Contribution

The expected contributions of this research are inferred from the research goals **RG-1**, **RG-2** and **RG-3**, and they are the following:

- *a formal model* which describes a heterogeneous computing platform with respect to the necessary resources for normal[5] execution of software components, availability of resources provided by computing units and constraints defined by the software architect,
- *a framework* for optimizing the allocation of software components on computing units in heterogeneous computing environment, and providing an insight into the performance of the future system in an early design phase,
- *a method* for the verification of the suggested model through the comparison of system performance predicted by the suggested model and extensive real world measurements,
- *a tool* which provides support for early-stage system design modeling and decision making.

These contributions are aligned with the research direction defined by researches involved in this domain of software engineering [58, 60, 111]. They address the coming challenges of complexity, heterogeneity, dependability and adaptability. The findings and conclusions presented in this thesis are applicable to distributed systems in automotive networked systems, avionics, manufacturing and embedded systems [111].

Hypothesis

The flow of the research presented in this thesis is guided by the following hypothesis:

H-1: *The suggested framework for comparison of individual software component allocations onto computing units correctly represents the system performance if the model for heterogeneous computing platform is substantially valid.*

H-2: *In a case with large number of allocation alternatives the developed algorithm for automated search for optimal allocation finds the allocation which has significantly*

[4]Having m computing units and n software components results in a design space of m^n (chapter 3).
[5]One conforming to a given specification of functional and extra–functional requirements.

better performance than random allocations.

The first hypothesis referrers to two main aspects of the subsequently suggested framework for allocating software components onto a heterogeneous computing platform. The first aspect relates the model for comparison of the heterogeneous system performance. This means that the system performance predicted by a model needs truly to reflect the system performance measured on a real–world system. It also suggests the procedure for testing this part of the hypothesis. The second aspect of hypothesis **H-1** refers to the content that such a model can contain, i.e. describe. This needs to be clearly specified and defined by the model so a software architect can know to what kind of systems this is applicable.

The second hypothesis considers the cases with large number of possible allocations a software architect can choose from. In some occasions this number can be overwhelming for an exhaustive search method. Therefore, an automated method for generating allocations should be designed. Depending on the design space size, it is acceptable to consider sub–optimal solutions. Considering that there are different methods for obtaining sub–optimal solutions, each of the methods proposed in this work should be benchmarked against each other to compare their performance. Furthermore, to emphasize the significance of the obtained result (i.e. allocation), the methods should be benchmarked against randomly generated allocations to prove that in a statistically significant number of occasions, the method(s) suggested in this work, generate allocations with statistically significantly better performance than the performance of the randomly generated ones (since there are no other existing models to which they could be compared to).

The research scope

Considering the presented research questions and objectives the work presented in this thesis classifies as a *software optimization research*, focusing on the emerging heterogeneous computing systems, i.e. systems which contain different types of computing units with different computing paradigms. Having this in mind, one can infer the question of who decides where does the software operate, i.e. execute? – With a commodity of different computing units, one can surely place every software component to the computing unit where it fits the most. This work presents the research about the best fit of the software within the heterogeneous computing platform considering the following scope:

– *static system analysis*, although the methods for optimizing software architecture presented in this work are not applicable explicitly only to the static analysis of the software architecture, they are verified in a static architecture environment. For the dynamic software allocation analysis some additional considerations should be made, hence the results of this research are bound to the static system analysis.

– *early stage design phase*, the software architecture optimization techniques presented here is envisioned, designed and verified for the software architecture in its early design stage with the goal of providing an insight into the future system performance.
– *extra–functional properties need to be quantifiable and measurable* – the input of the decision model to find the best fitting software architecture (allocation) can account for unlimited number extra–functional properties, they are related to the system performance, can be quantified, measurable or carefully estimated.
– *the components are composed in a pipe–line architectural style* – the method for finding the optimal software architecture suggested by this work does not conform to any particular software architecture or architectural style, however the verification is made on the architecture with a pipe–lined architectural style. This means that the software components are executed with a certain order defined by different execution scenarios. Furthermore, the granularity of software components is arbitrary and should be defined by the software architect.
– *the model allows some approximations*, although all the main inputs for the allocation decision model are exactly measured, in the interest of model simplification and abstraction, it does allow some approximations.
– *measurements are performed with confidence interval of 95%*, since the confidence index of 95% represents statistically significant results and it was possible to perform it with the available equipment, this value was selected as satisfactory.

The in detail classification of this research within the subject area of software architecture optimization can be found in section 6.1.

1.4 Research method

The research method followed in this work is shown in Figure 1.1. It is a fusion of good practices for performing software engineering research suggested by Mary Shaw[104] and a general research method suggested by Richard Feynman[39]. The presented method is custom developed for this research, however it is fairly general and applicable to other software engineering research within the domain.

The research method consists of three distinctive repeatable phases. In the first phase (the problem definition phase) one needs to identify and formulate the research problem using current scientific knowledge. This refers to systematic study of related publications available in relevant conference proceedings and journals. Once the problem is identified and formulated it needs to be grounded to a specific context. It needs to be very clear and precise about the context of the problem and the assumptions under which the suggested solutions/answers/explanations work. From this knowledge one can derive specific research goals which lead to concrete research questions. In this research, the type of research goals imposes a requirement for verification of the

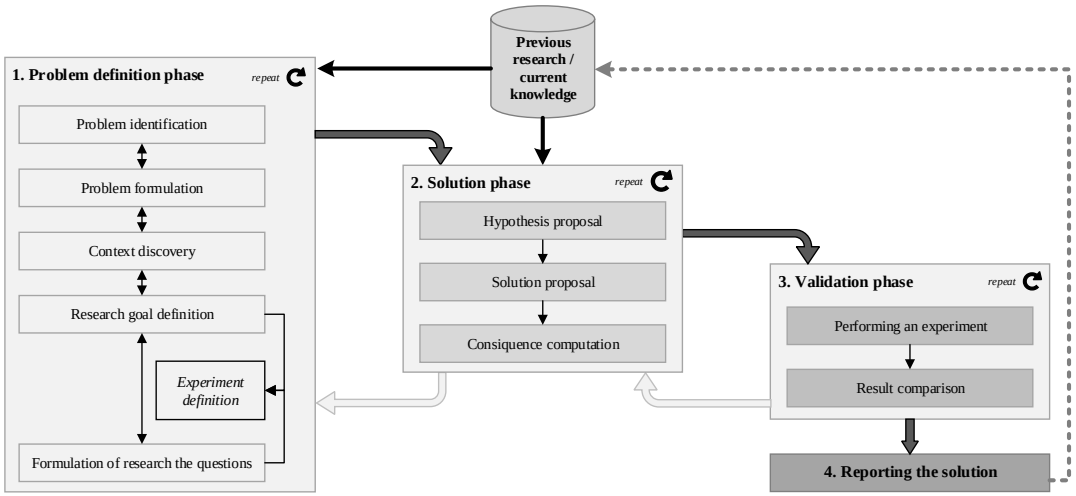

Figure 1.1: Three-phase research method developed and used in this thesis.

solution to gain trustworthiness of the suggested model. Therefore, in the first phase it is also necessary to define the parameters for the verification of the proposed solution, i.e. the experiment.

In the second phase (the solution phase) using current insights and known principles one needs to propose a solution to previously defined research question(s). The proposed solution must not be vague. It must be clear and unambiguous because the next step involves *making an assumption*, i.e. a *prediction* by the suggested model which describes a phenomenon.

In the final stage (the verification stage), through experimentation is made by which one verifies if the calculation results, i.e. the model prediction matches the measured results obtained on a real–world system. If the experiment and the calculation do not mach, the proposed solution is wrong, hence, it is necessary to repeat one of the previous phases, as shown in Figure 1.1. If the results of the experiment (repeated multiple times) mach with the calculations, the solution is valid, which brings us to the last optional step. When solution to the problem is found and verified, in scientific community researchers report their findings through journal and conference papers, thus updating the existing knowledge base.

1.5 Thesis outline

The rest of the thesis is organized as follows. *Chapter 2 – Background* presents the state–of–the–art research which motivated this thesis. It also briefly presents the scientific discipline of software engineering, its historic development, along with the fundamental facts about CPUs, GPUs and FPGAs necessary to follow and understand

the research context of this thesis.
Chapter 3 – Mathematical Model of a Heterogeneous Computing Platform introduces the mathematical model for the formal description of a set of heterogeneous computing units, a set of software components, a set of constraints and a cost function which quantifies the quality of an allocation. *Chapter 4 – Measurement* presents the collection process and an analysis of extra–functional properties for a component based software architecture deployable on different computing units within a heterogeneous computing environment. *Chapter 5 – I-IV Framework Verification* shows the verification of the suggested the I-IV framework. *Chapter 6 – Related work* presents the work of other researchers closely related to this thesis, closely concerned to software architecture optimization and cyber–physical systems. And the final *Chapter 7 – Conclusion* concludes the research, presents the answers to the research questions and outlines the future work.

CHAPTER 2

BACKGROUND

This chapter presents the state–of–the–art research which motivated this thesis. It also briefly presents the scientific discipline of software engineering, its brief historic development along with the justification and placement of this work within the component based software engineering research. The related work presented in chapter is a broad review of the field which includes, software modeling, software architecture, heterogeneous computing systems, etc. Furthermore this chapter provides an overview of the processing units (CPU, GPU, and FPGA) used in this research, along with their fundamental design principles necessary to follow and understand the research context of this thesis.

In last several years the increase of central processing unit (CPU) clock came to a halt, but despite that the performance of computers is still growing. Although it used to be a popular measure of processing power, the CPU clock frequency today hardly has any mention. Since increasing the CPU clock frequency proved to be inefficient[1] there has been a paradigm shift in the CPU architecture. Modern CPUs feature multiple cores which enable real parallel processing. Depending on the task type, multicore processors handle some tasks very well, however with some tasks they struggle. So, the current processing platforms also have dedicated processing units specialized for certain tasks[84]. Having multiple types of computing units led to a new era in computer science referred to as *heterogeneous computing*. The research presented in this work focuses on such systems which in particular consist of a central processing unit, graphical processing unit (GPU) and field programmable gate arrays (FPGA).

Due to the parallel nature of graphics processing, GPUs evolved into high performance systems capable of running thousands of threads simultaneously, spread across hundredths of GPU cores [49]. In comparison to CPUs which typically have a lot less processing cores, GPUs are superior however, for some tasks CPUs performs better while for others GPUs do. The main difference is in the type of the task which they are designed to handle. While CPUs are better in handling parallel tasks which involve a lot

[1] Related to energy consumption, thermal issues and caching [24].

of branching i.e. decision makin. However, handling tasks without branching (e.g. single instruction, multiple data - SIMD operations) is where GPUs excel and outperform CPUs. As good as GPUs are for massive parallelism, there is a large drawback related to high energy consumption which makes them undesirable for embedded systems. In embedded systems, along with functional requirements, extra–functional properties (EFP) are equally important. EFPs describe the quality attributes and ensure the proper behavior of a system. Usually they depend on characteristics of the underlying platform and the architecture of the system, however some examples include reliability, latency, real-time response, bandwidth, adaptability, accuracy, power efficiency etc. Therefore, with regard to power consumption in particular, in embedded system designs GPUs are often omitted by a more energy efficient alternative; field programmable gate array (FPGA). FPGA is essentially an *empty* chip which is made to be *programmed* for any for any custom functionality. Despite the advantages of FPGAs over GPUs (lower energy consumption by the factor of 10 times [42, 86] and physical size) the development effort for even basic functionality is much greater than for GPUs. Still in some cases[2] GPUs tend to outperform FPGAs [42, 44, 55, 86].

It is not likely that GPUs will replace CPUs, and FPGAs will replace GPUs in embedded systems any time in the near future. Each of these computing units has advantages and disadvantages. They complement each other so joining them in a single heterogeneous computing platform provides new possibilities for advancement in computing.

CPUs, GPUs and FPGAs come from different backgrounds and therefore have different programming paradigms. Developing software for such heterogeneous systems is particularly hard since developers now need to handle more details, e.g. task types (sequential or parallel) communication bandwidth between computing units, data type conversions, energy consumption, development effort, pricing, etc. This work focuses on the *software engineering* challenges related to exploiting the benefits of CPUs, GPUs and FPGAs available on a single heterogeneous computing platform and the architectural decision making concerned with placing software components to the best possible computing unit.

2.1 Software engineering

2.1.1 The golden age

It has been bore than twenty years since Mary Shaw [103] famously questioned the *engineering* of software engineering and set some foundations to bring it closer to other engineering disciplines. Since then, there have been some remarkable advancements of

[2]Processing with less memory exchange between CPU and GPU; e.g. optical flow, detecting patterns on large images, multiplying large matrices, N-body simulation, etc. FPGAs perform better for stream processing and pipelines.

the field, so in 2006 along with Paul Clements, Mary Shaw reported on the *state of the software discipline*, namely focusing on software architecture [105]. It was suggested that it typically takes 15 to 20 years for a technology to become ready for popularization. How does this reflect on software architecture?
Comparing the rate at which results of previous work are used as a building block for the subsequent research, software architecture matured and reached it's *golden age*. Also, there are other maturity indicators, such as: industrial trainings, certification programs for software architecture, standards, patterns and tactics, evaluation and validation methods, tools, journals, etc. The challenges for future software architecture researchers in 2006 were mostly related to quality, formalization, modeling notation and languages, assuring conformance of model and code, organizing architectural knowledge and adaptation of software systems to changes.

In 2009, the paper about golden age of software architecture was revised [22]. Clements and Shaw reported that there are some strong indications that software architecture is enjoying popularization, however they pointed out that "*architecture will always be architecture*", primarily thinking of new computing platforms which demand new architecture styles and the possibility of a never ending loop, where new emerging computing paradigms require new software architecture principles, *ad infinum*. However, ones which get to be old enough, get to see the same things repeating, but in the different form.

2.1.2 Expanding body of knowledge for software engineers

Since appearance of new computing platforms demands new architectural styles, software engineers need to broaden their design considerations. IEEE Computer Society recognized this by expanding the knowledge areas for software engineers. In their latest edition of SWEBOK v3.0[3], they included new topics for software design; architectural decisions, software design tools, security and user interface design. In addition they included new knowledge areas: economics, computing, mathematical and engineering foundations [52].

Similar conclusionas are made in the literature review by Teich [111]. He pointed out that software engineering overlaps with the hardware design discipline and that the *hardware-software codesign* techniques should be known to anyone who wants to keep up with challenges of increasingly complex electronic system designs. This goes for SoC[4] designers, software and hardware engineers, distributed systems in automotive, networked systems, avionics, manufacturing, embedded systems. In this field tool–supported specification, modeling, partitioning and synthesis of subsystems is of utmost

[3]Software engineering body of knowledge – an international standard which defines knowledge areas for software engineering discipline

[4]SoC - system on a chip is an integrated circuit which integrates all the components of a computer into a single chip

importance in order to be able to build complex systems with tight extra-functional constraints such as cost, performance, power, temperature, reliability and time [116].

2.2 Component based software engineering

In order to handle the increasing complexity and to provide functional cost–effective solutions, software researchers and practitioners often use component–based technologies/principles. Instead developing all parts of the system from scratch, reusability and modularity are the main concerns. Component based software engineering (CBSE) is a branch of software engineering which deals with models, methods and frameworks for development of component-based systems. The main principle of CBSE is separation of concerns into smaller building blocks called components. Components are: a) easily replaceable, b) units of deployment and c) without observable state [110]. For this work, the most suitable definition is one by Szysperski:

"*A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to third-party composition*" [109].

According to his definition, the function of a software component is independent of the system on which it is deployed. Moreover, software components are independent of each other and they communicate through well defined interfaces. This makes components easily replaceable without the apparent affect to other components and to the rest of the system.
CBSE principles provide a solid foundation for dealing with software engineering challenges related to heterogeneous computing platforms. Managing different computing unit types, different data types, different processing rates, programming paradigms and EFPs can be simplified by separating concerns in to smaller manageable units, i.e. components. Each component would handle its own tasks, could be allocated to a dedicated computing unit and be accessed via predefined interface. In order to design a software system, an architect would choose components from a repository and join them together into a complex system.

In the following section, presents the current main concerns for CBSE researchers and explores the possibilities of applying CBSE principles to software design for heterogeneous computing platforms.

2.3 Hot topics in software engineering research

To gain better understanding of currently popular topics in component based software engineering and high performance embedded systems a simple three step analysis was performed.

The first step involved discovering the main research issues and to get acquainted with specific terms by seeking for related publications in *IEEE Xplore* database. The search term *software components*, limited to last few years (2012-2015) resulted in more than 4000 publications. This result was imported into a spreadsheet application for further analysis. Next, the results were filtered to include only papers published in journals and conferences which contain words *'software'* and *'comput*'* in their name. Further filtering limited only publications which contain the word *component* in their title. This resulted in 137 publications. Abstracts of these publications were inserted into a text analyzer software[5] to find most frequent words (excluding the words *component*, *based* and *software*). According to this, most of publications report on research in: a) *reuse*, b) *frameworks*, c) *quality*, d) *architecture*, e) *reliability* and f) *performance*.

The second step of searching for hot topics was to find most common subjects of papers published in most influential journals[6]: *IEEE Software*, *IEEE Transactions on Software Engineering*, *Journal of Information and Software Technology*, *Journal of Systems and Software*, *ACM Transactions on Software Engineering and Methodology*. Most common topics were on a) *reliability and testing* [14, 29, 50, 50], b) *software models, system development and analysis* [45, 61, 66, 83, 85, 119, 125], c) *principles and CBSE methods* [27, 30, 31, 76, 92] and d) *decision making in CBSE* [11, 28].

By using the keywords from the lists of common research subjects obtained in previous two steps, a full state–of–the–art search was conducted which included the following databases: *IEEE Xplore*, *ACM*, *Scopus*, *Web of Science* and *Google Scholar*. The resulting papers were classified in five classes:

- Software modeling for embedded systems
- Software architecture for embedded systems
- Modeling heterogeneous platforms
- CPU, GPU, FPGA systems
- Software component allocation

All the publications within these classes will be presented in the following subsections. These publications represent the state–of–the–art in component–based technologies and high–performance embedded systems. They were explored in order to discover the research opportunities and guidelines for future work as suggested by other scientists involved in this field.

[5] http://textalyser.net/

[6] Journals were chosen based on experience, however following two sources justify the selection of journals: http://www.cse.chalmers.se/~feldt/advice/isi_listed_se_journals.html, http://academic.research.microsoft.com/RankList?entitytype=4&topDomainID=2

2.3.1 Software modeling for embedded systems

To better understand and communicate ideas about the code which runs computer systems researchers and practitioners use software models. Software modeling is a popular research topic and most of the current publications are focused on simulation of hardware or software systems [38], testing [56], model verification and formalization [17]. Within modeling community, model driven development (MDD) is a particularly hot topic. Although MDD is not a new idea, it is particularly interesting for embedded systems researchers because it handles complexity well and minimizes human made errors, thus increasing the reliability of a system. A lot of publications focus on creating models for industry verticals (i.e. DSL - domain specific languages) and on redefining, refining, extending current models. To make them trustworthy, researchers often deliver a demonstrators or a use–cases. Martinez et.al. argue that code–centric embedded software development for embedded systems is complementary (and slowly fading) to the new pervasive model–centric approach [73]. Khune et.al. bring this to the next level by using model–centric approach which generates complete and executable code [115].

Creating new models and software modeling techniques are the most common suggestion for future researchers. It is emphasized to focus on modeling of particular system properties and their representation and interpretation within the model [117]. Although software modeling reached the point of synthesizing executable code from models, one also needs to consider higher levels of abstraction, i.e. the architecture of the entire software product.

2.3.2 Software architecture for embedded systems

Along with software modeling, its architecture is a common subject within the modeling community. Many researchers use a modeling languages to create software architecture of a specific system and observe the system behavior through the model via simulations. Through such observations, new system behavior and properties can be discovered and used to form architectural (anti–) patterns.

Using the concepts of meta–architectures in architectural description languages (ADL) Adel and Abbdellah created a meta–ontology which enables semantic mapping from ADL to model driven architecture (MDA) platform [2]. This work is significant because it creates a connection between software architecture and the platform it implements. Although it a good model for bridging software architecture with the implementation platform it is not applicable to heterogeneous systems. A work by Hause and Thom is very similar [47]. They integrated MDA with SysML[7] and UML[8] to provide a holistic view on requirements, concurrency, physical and logical architecture. Their main goal

[7] System modeling language – http://sysml.org/
[8] Unified modeling language – http://www.omg.org/spec/UML/

was reusability. Similar work was done for multicore software development by Chihh-siong et.al. In their paper [106] the focus is on a framework development for multicore domain which enables automatic code generation from SysML models. Multicore has also been topic of interest to El Marabti et.al. who presented language for modeling application architecture and code generation flow for a multiple processor platform [77]. Once the architecture is well defined and the semantics of the model is clear, some architectural patterns can be found. More on this can be found in the work by Sandrieser et.al. who focused on heterogeneous platforms [99].

Many scientists mention heterogeneous platforms in their papers and it seems to be a popular research subject, however only a few publications really focus on heterogeneous platforms which contain CPUs, GPUs and FPGAs. Modeling of heterogeneous platforms is hardly a primary research problem of papers in this category. Very often, heterogeneous platforms are related to verification or demonstrator of the publication. This is further discussed in the following subsection.

2.3.3 Modeling heterogeneous platforms

An interesting research related to model driven development for embedded systems has been conducted at Mälaralen University under the project PRIDE. A work by Brode et.al. [12] and Calrson [18] presents an environment dedicated to the development of distributed embedded systems using the ProCom [15] model. The approach is interesting and novel because it implements an incremental approach to synthesizing runnable representation of model entities. This largely contributes to simplification of complex systems.

Next very common topic are GPU programming models. These models opened a new path for exploiting massive parallelism of GPUs. Some authors are more hardware oriented [7, 36, 113], and other are concerned with model driven engineering (MDE). Since the primary interest of this thesis is on the software, one publication which stands out is by Jablin et.al. Authors address the limitation of GPU-CPU communication and present a fully automatic system for managing and optimizing this communication [54].

A paper [95] by Rodrigues et.al. addresses the development for GPU platforms which uses MDE and MARTE [82]. The outcome is hybrid–meta model in UML MARTE which can generate compilable OpenCL code. However there is a problem with MARTE, SysML and AADL. These modeling languages can express the same concept in different terms. Ziani et.al. addressed this issue with their own meta–model, RCES, which is a unified meta–model for resource constrained embedded systems [126]. Also there are timing related issues with AADL and MARTE which are addressed by Mallet et.al. [69].

This research area is relatively recent and there are a lot of calls for papers on these subjects within software modeling community. Despite the existence of modeling lan-

guages, researchers tend to avoid them in favor of their own proposals[9] mostly because existing languages are not explicit or expressive enough. Also heterogeneous computing mostly refers to combinations of CPU and GPU while other computing units are neglected. The further investigation on this issue is presented in next subsection.

2.3.4 CPU, GPU, FPGA systems

Since heterogeneous platforms host computing units of different types, such as CPUs, GPUs, and FPGAs, tasks which run on these units can be handled at very different rates. This is due to communication latency or processing speed. Most of the researchers currently report on hardware and software issues related to handling systems with CPUs, GPUs and FPGAs and performance evaluation for specific tasks.

A work [101] by Senouci et.al. addresses the difficulty of bridging communication between processors in heterogeneous multiprocessor systems. Proposition of their work is to use FPGA based middleware which would handle the *glue code* to bind all the components together. Importance of middleware is also recognized by Ibrahim et.al. who proposed message oriented middleware as a glue technology for heterogeneous distributed system and also loosely coupled software systems [51]. Yang-Hsin et.al. proposed software synthesis for middleware which can automatically generate software for heterogeneous embedded systems [35]. Similar work is done by Chouhan et.al. [21]. In their paper authors address the problem of performing deployment for the heterogeneous platform using deployment heuristics. Xia and Chen focused on hardware and presented a component based framework for embedded digital instruments [124] . Authors point out that design of such systems should follow truly pattern–driven software development methodology to achieve full reusability. In order to abstract both hardware and software components new methods are needed which would enable independent architecture regardless of implementation and location. Rincon et.al. addressed this issue by defining a low overhead, system–wide communication architecture that offers complete communication transparency [93]. Article [13] by Brinkschulte et.al. focuses on providing a suitable communication platform for real-time applications in a heterogeneous environment in the form of middleware called OSA+. Also, some authors address these issues by proposing component oriented approaches [65, 91].

Many papers also report on performance comparison between a CPU, a GPU and an FPGA [9, 44, 55, 86]. It is obvious that different computing units excel at different tasks. Careful allocation of tasks can result in best performance. Therefore modeling of such system is an important step toward performance estimation in early design phase. Similar observation is also reported by Senouci et.al. [101]. Since allocation of different tasks can result in different performance, this is an interesting point for CBSE researchers.

[9] More on this topic can be found in literature overview by Aleti [3]

2.3.5 Software component allocation

Due to high dimensionality of modern enterprise systems and adaptive software systems which emerge from ubiquitous computing, some authors [123] argue that there is a lack of formal methods to cope with the complexity. In the current literature some authors [46, 87] consider that CBSE is a solution for supporting adaptation and complexity reduction by enabling third parties to independently develop, deploy and compose parts of the solution. In their paper, Massow et.al. used CBSE principles to show the architectural runtime reconfiguration in distributed software systems using Palladio Component Model using performance simulator called SLAstic.SIM [75].
Another research trend focuses on deployment alternatives. Lombardi et.al. [32] propose an extension of IEEE Recommended Practice for High Level Architecture Federation Development and Execution Process by providing an approach for optimizing the allocation at high level. Their work resulted in a tool called S–IDE which provides feasible deployment alternatives, however it lacks addressing multiple extra–functional requirements. In [19] authors present an allocation method to improve resource utilization and scheduling while keeping the decisions independent of platform implementation. The method determines the allocation using a genetic algorithm with an optimization function (heuristic algorithms [107] are often used since allocation is a NP-hard problem [53]).
Each deployment scheme may impose two kinds of delay on the overall performance, communication due to remote invocations and computational due to resource sharing [100]. Considering this, some authors focus on the bandwidth [19] while other focus on computational delay [100, 118] using different optimization methods to find the best solution.
Another important focus is brought forth by Bushehrian et. al. in [16]. According to their experience, multiple resource constraints can sometimes be inter-dependent and conflicting. Therefore the system must be viewed as a whole and its properties need to be measured and modeled accurately because it can have great impact on the trustworthiness of final result.

Most of the related work focuses on allocating components, optimizing performance and utilizing some mathematical method to model and solve the problem. However, *heterogeneous systems* and *multi-objective* design decision making needs to be further investigated due to the presented issues. These are mostly related to the following challenges; solving compatibility issues of different computing paradigms in heterogeneous computing, finding efficient methods for dealing with high dimensionality of problems related to software deployment in such systems, deal with design decisions in a systematic and formal manor, deal with methods for quantifying extra–functional properties in heterogeneous systems, increasing the time necessary to effectively design efficient systems, etc. These challenges are the integrate part of the research presented in this work.

More of the closely related work relevant to the research subject of this thesis is presented in detail in chapter 6.

2.4 Heterogeneous computing

The definition of heterogeneous computing is hardly ever mentioned in scientific writing and publications. It almost used as a phrase or as a name for the type of computer, rather than a strictly defined term. Generally heterogeneous computing refers to computing systems which use more than one kind of processor[10]. However, a strict definition by Gaster et.al. also provides an explanation for its purpose; *heterogeneous systems are assembled from different subsystems, each of them optimized to achieve different optimization points or to address different workloads* [43]. Slightly older definition presents heterogeneous computing *as well–orchestrated and coordinated effective use of a suit of diverse high–performance machines to provide super–speed processing for computationally demanding tasks with diverse computing needs* [1]. In this research, the working definition will be the combination of previous two: heterogeneous computing refers *complex systems composed of different kinds of processing units which use different processing paradigms and are designed for different types of tasks which work together in order to provide the best processing performance for diverse computing needs.* While the term *computing platform* will be used, *in the most general sense, as whatever preexisting environment a piece of software is designed to run within, obeying its constraints, and making use of its facilities*[1]. Since the main focus of this research are systems which contain multiple computing resources of different types, the previous definition will be amended with the term *heterogeneous* to become a *heterogeneous computing platform*. As such, the definition remains the same but the preexisting environment refers to environment with many computing units of different types, and also includes many software components which can be allocated onto these.

Present heterogeneous computing platforms typically contain software components which can be assigned for execution among many different computing units such as CPUs, GPUs, FPGAs, DSPs, ASICs, ASIPs[11], etc. For this work the first three of these are the most important.

2.4.1 Central processing unit

The central processing unit (CPU) is often referred to as the brain of a computer. It is rightly so, because it manages and coordinates all the components of a computer. It consists of arithmetic logic unit, control unit, set of registers, clock and several levels

[10]AMD Heterogeneous Computing Developer Central - http://developer.amd.com/tools-and-sdks/heterogeneous-computing/ (accessed: 29.3.2015)

[11]DSP – digital signal processor, ASIC – application specific integrated circuit, ASIP – application specific instruction sec processor

of cache memory. CPU can be described on several different viewpoints, from low-level transistors to a bit higher level which consist of logic units. This work will briefly present the low level, all the basic constituent units of a CPU and focus on the instruction set and processing type.

Motherboard, which connects all the components of the computer connects CPU with random access memory, i.e. RAM. RAM usually stores three types of data; instructions, numbers (data), addresses. Usually CPU reads this data in order and processes it, however it can also skip over some data and continue from different location in the memory, hence *random access*. The control unit (CU) inside the CPU receives instructions form RAM and processes it further, usually in the arithmetic logic unit (ALU) which commonly implements electronic circuits for addition, subtraction, comparison etc. Temporary data, i.e. operands and results are stored in registers and transferred between internal CPU components using a CPU bus. All the operations which occur within a CPU are directed by the CPU clock, which synchronizes both internal and external communication. A time required to perform one such operation is called a CPU cycle and it directly influences the processing speed.

Since the CPU is several orders of magnitude faster than RAM, data transferred between CPU registers and RAM is stored in specialized memory to alleviate possible bottlenecks, i.e. caches. Level one cache (L1) stores data which is instantly available to the CPU, and it is measured in kilobytes while level 2 cache (L2) is slightly slower, typically outside CPU and bigger, it can store several megabytes. In present multicore designs, L1 and L2 cache memory can be typically private for processing cores, while another cache memory level (L3) is added to be shared among the processing cores on a single chip.

Typical instructions which are stored in memory are `LOAD`, `STORE`, `ADD`, `COMPARE`, `JUMP IF`, `OUT`, `IN` etc. These instructions are used in assembly language to develop machine programs. The available instructions are bound to the CPU architecture which is defined by the instruction set architecture (ISA). The design of the CPU determines the ISA which can be implemented on a particular chip. Instruction set architecture is well defined, documented and standardized interface, i.e. contract between hardware and software. It defines a functional definition of operations, modes and storage locations supported by hardware and a precise description of how to invoke and access them. And good ISA is defined by its pragrammability, implementability and compatibility features [72].

For this work, the most interesting CPU designs[12] are CISC (complex instruction set computing) and RISC (reduced instruction set computing) since the most widely used ISAs belong to these families, i.e. x86-64 to the former and ARM to the latter.

The main idea which guides CISC CPU design approach is to complete a task in as few lines of assembly code as possible, and therefore it implements complex instructions

[12]More on this subject can be found in [48]

which don't need explicit calls of loading or storing functions. CISC also calls for using complex multi-clock instructions, using less code, high cycles per second etc. [20].

RISC on the other hand, uses a different philosophy. It involves using single-clock simple instructions, use registers to load data for instructions, low cycles per second and large code sizes. Since each clock cycle is used to execute one instruction the entire program executes in the roughly the same time as multi–cycle instruction usage. In fact, in the CISC approach, on some level instructions are translated into RISC-like operations [20, 48].

x86/x64

x86, also known as IA32 belongs to the CISC family and is traditionally more powerful but less energy efficient. x86 dominant instruction format in world's computers running on Windows, Linux and since 2007 MacOS X. This instruction set architecture was defined in 1985 with the introduction of Intel 80386 microprocessor which extended the original 8086 architecture. This architecture is fading away in favor of x86-64 architecture, originally developed by AMD which is a 64-bit extension of x86 architecture [10]. This basically means that the processor can address a memory space of 2^{64} bytes.

ARM

During the 80's when CPU design took a greater momentum and appearance of Apple Lisa with 16-bit processor, British company Acorn realized that existing 8-bit machines should be upgraded to increase performance of future systems. Since they already used RISC, their goal was to develop a high performance RISC processor, and therefore Acorn RISC Machine, i.e. ARM[13] was born in 1985 [64]. ARM belongs to RISC family and it's typically less powerful than x86, but more energy efficient, hence it is majorly used in embedded systems.

Currently, with the growth of the marketplace for tablets, smartphones and other *smart* devices running ARM processors, they enjoy a great popularity and they surpassed the market of desktops and laptops using x86 ISA. Also, there is an interesting market phenomenon happening today; traditionally low–power ARM is entering high–performance server market, and traditionally high-performance x86 is entering low-power embedded market. The recent research shows that today, RISC vs. CISC, i.e. ISA is irrelevant for power and performance characteristics (at least for ARMv8 and higher)[10] since present technology made them almost equally efficient.

[13]Later the acronym way changed to Advanced RISC Machine

Multicore CPU

The processing speed is largely dependent on the CPU cycle duration. If the duration of the cycle is reduced, more operations can be executed in the same amount of time. Therefore during the 90's and early 2000's, speed–ups in software largely relied on decrease of CPU cycle duration, i.e. higher operating frequency. However, due to the current design of transistors there is a theoretical limit to how small a cycle can be before CPU becomes inefficient (memory wall, ILP wall, power wall [48]).

To overcome the inefficiency of CPU design due to physical limitation, the current design trend relies on replicating processing units, or more commonly refereed as processing *cores*, within a single chip. This allows parallel execution of programs, i.e. threads (independently manageable instructions), but also ended the *La-Z-Boy* era[14]. Therefore if programmers want to make their program faster, with each generation of computers, they need to make their programs more parallel, hence the Moores law, according to Hennesy and Patterson is up to programmers. The switch from to multiple processors per microprocessor led to the term core which is also used as a processor. Instead multiprocessor microprocessor, the term multicore caught on [48]. The present day chip production process enables the development of CPUs with a lot of cores on a single chip with low energy consumption and shared internal memory (L3 cache). A typical design is shown in Figure 2.1.

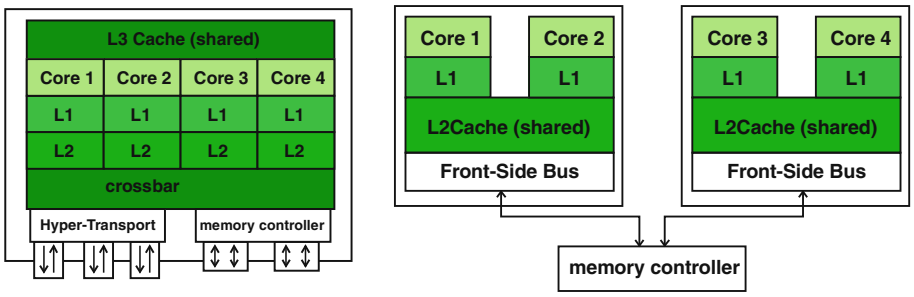

Figure 2.1: CPU Architecture, left image shows a hierarchical design multicore processor where cores share L3, and have their own L1 and L2 (AMD), while the right figure shows a design where two cores share L2, and communicate via a memory controller (Intel) (from [90]).

In order to utilize multiple cores, on higher level often different libraries are used, such as *OpenMP* or *OpenCL*. Currently modern compilers perform some compiler–level optimizations and offer a programer set of libraries to be used for parallel programming (e.g. `Parallel` in C#, `multiprocessing` in Python or `Concurrent` in Java). It can be said that today it has became fairly common.

[14] refers to the present day programming problem where computing performance is programmers burden, and the programs cannot run faster without an intervention from a programmer as it was in the past

2.4.2 Graphical processing unit

Previous section presented the current trend in processor design which obviously exploits parallel execution of programs. Since this is a key advantage of GPUs, in order to understand theri design it is necessary to understand parallelism in computing. According to well known Flynn's taxonomy of data–level and task–level parallelism, the two most important groups for this work are: *single instruction stream, single data stream* (SISD) and *single instruction stream, multiple data streams* (SIMD). SISD category is a standard *uniprocessor* for which a programmer writes sequential code. On the other hand SIMD category applies the same operation to multiple items of data. The same instruction is executed by multiple processors. SIMD is a characteristic of GPUs, although there are SIMD ISA extensions for CPU's (e.g. MMX, SSE, AVX) [48].

The GPUs are primarily made to handle processing of graphical models. These are usually represented as triangles and any graphical transformation or rendering for the user is made by applying a *pipeline of operations*[15], e.g. lighting, camera simulation, rasterization, texturing, etc. These operations are often called shading tasks, and therefore computing units and / or programs which can perform shading tasks are called shaders. Historically, these were dedicated hardware elements. Since this made graphical hardware hardwired, by design and necessity they evolved into a programmable computing units called unified shader. This introduced GPUs with increased flexibility and added support for longer programs, more registers, and flow-control primitives [67]. Unified shader architecture provides one large grid of data–parallel floating–point processors general enough to run different workloads which soon enough led to general purpose graphics programming units, i.e. GPGPUs.

The highly parallel workload of real-time computer graphics demands high arithmetic throughput, however tolerates a considerable latency, because images need to be displayed only every 16 milliseconds (for 60FPS graphics). Where CPU is optimized for low latency, GPUs are designed for high throughput[67]. For a pipelined design, GPUs load data elements and process them in multiple execution cores. Data elements enter the processor chip via an input port and successively flow through different cores until the processed data elements leave the last core and the entire processor chip [90]. GPUs have a different architectural design point than CPUs mainly focusing on efficient execution of parallel threads. This is apparent in the per-chip transistor budget where GPU spends more on computation and less on on–chip cache and overhead[80].

Figure 2.2 shows a typical GPU processor array which contains many processor cores. Depending on the manufacturer there are different terms associated for processor organizational units. It shows 128 *streaming processors* (SP) organized into 8 *multithreaded streaming multiprocessors* (SM). Each SP core is highly multithreaded capable of tens of concurrent threads [16]. The processors connect with DRAM partitions via an inter-

[15] A series of sequentially executed operations.
[16] nVidia Tesla architecture has SPs with 96 concurrent threads[80]

connection network[80]. SPs within each SM share a common texturing address unit, texture filtering unit and L1, L2 cache [81].

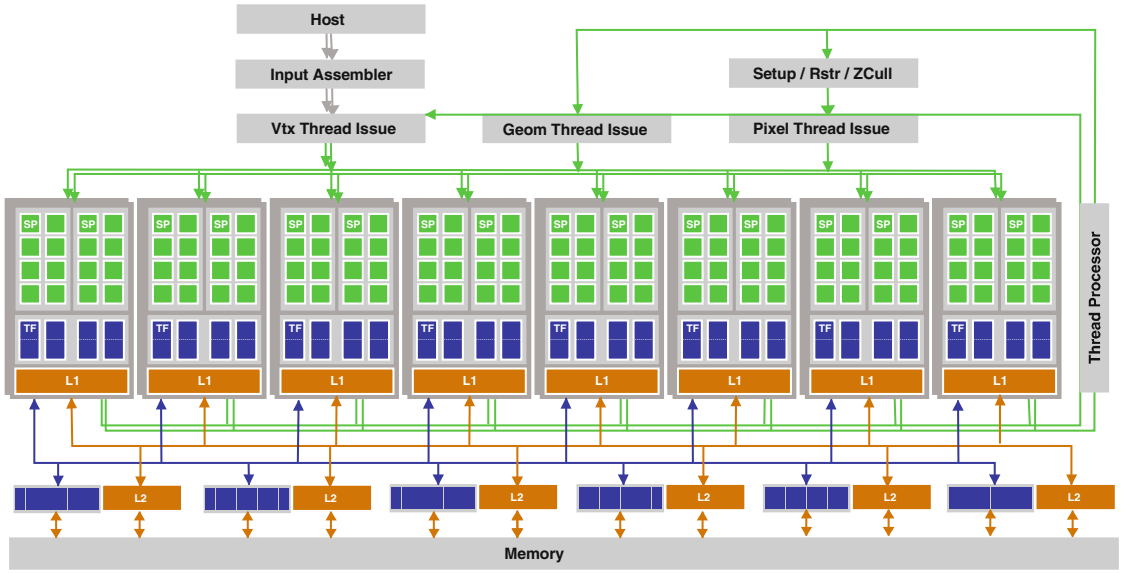

Figure 2.2: GPU Architecture of nVidia GeForce 8800 (from [90]). SP is a streaming processor, TF is a texture filtering unit, TA is a texture address unit.

Programming multiprocessor GPUs is different than programming multicore CPUs. As seen, GPUs provide several orders of magnitude larger parallelism with a continued increase over the years. Therefore a scalable programming model is necessary to simplify general purpose parallel computing and enable a programmer to write a code for a single thread and programs thus scale transparently over a wide range of hardware parallelism [80]. There are two mainstream programming models used for general purpose GPU software development, OpenCL and CUDA. OpenCL is open source and it is maintained by Khronos group, while CUDA is Nvidia specific. In this work, OpenCL was used.

Programming model – OpenCL

OpenCL (Open Computing Library / Language) is an open and royalty–free parallel computing API designed to enable GPUs and other task accelerating units to work in tandem with a CPU, providing it additional raw computing power [4]. OpenCL enables a transparent execution of programs with little regard of the underlying platform thus increasing a cross–vendor software portability. It binds the low–level layer and draws an explicit line between hardware and software. Upper–level software programmers cannot see hardware specific implementations such as drivers and runtime. OpenCL consists of tree main parts [4]:

1) *a language specification*, which describes the syntax for writing programs which run on supported accelerator units. These are referred to as kernels and they are based on ISO C99 specification with some extensions and restrictions,
2) *a platform layer API*, which provides a developer with an access to software application routines which query the system for the existence of OpenCL-supported devices, and
3) *a runtime API*, which provides *contexts* to manage multiple OpenCL devices, i.e. command queues, memory objects, kernel objects, etc.

Figure 2.3 shows OpenCL platform model which consists of one hosts and several *compute devices*. Each compute device contains *compute units* which are divided in multiple *processing elements*. Considering the Figure 2.2, it represents a compute device, each SM represents a compute unit and each SP represents a processing element. Processing elements execute instructions as SIMD (single instruction, multiple data) or SPMD (single program, multiple data). .

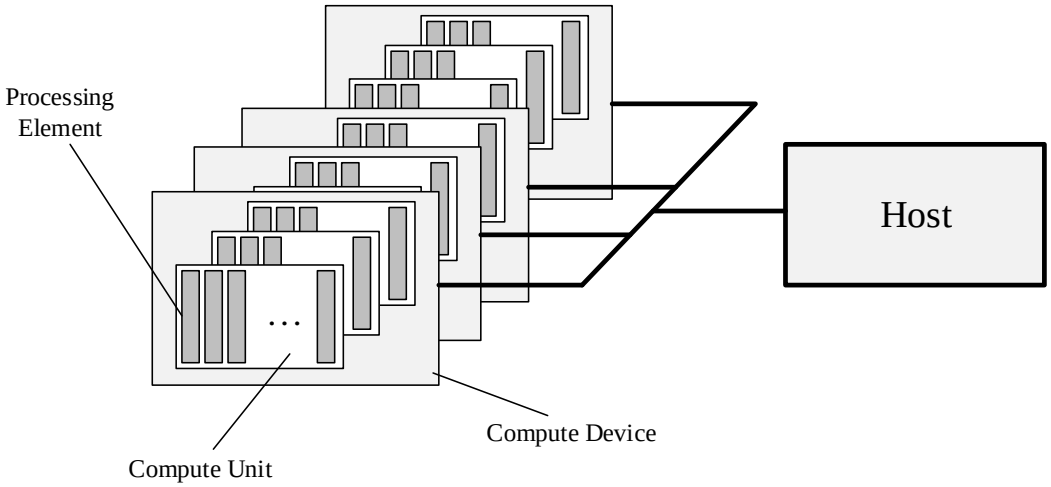

Figure 2.3: OpenCL platform model

OpenCL programs consist of two parts, *host code* and *device code*. Host code is written ina a general purpose programming languages, mostly C/C++ and executes on the host (Figure 2.3). From a host code, a programmer calls the operations available in the device code, i.e. kernel code which is written in OpenCL C and executes directly on OpenCL devices. Kernel code is a unit of execution which performs a function that can be executed in parallel. Parallelism is exploited by dividing a problem into an n–dimensional index space. Each processing element in the index space is called a *work item* and each one executes the same kernel function, but loads different data. n–dimensional index space can be either 1 (array), 2 (image) and 3 (volume) dimensional as shown in Figure 2.4.

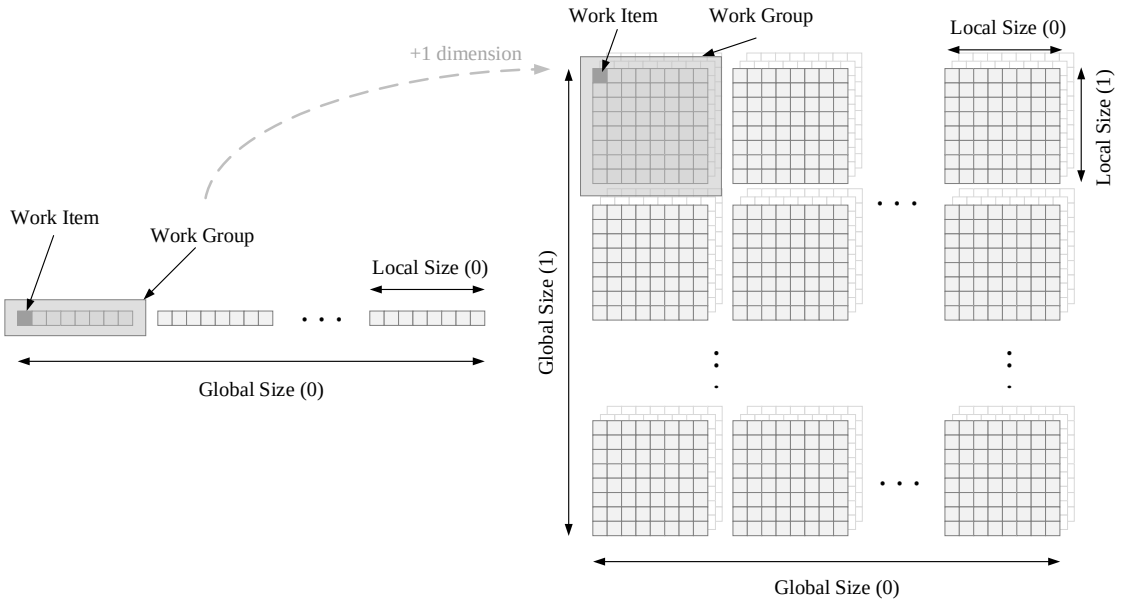

Figure 2.4: OpenCL Data–Parallelism, left side of the image shows data-parallelism operationg over an array, while the right part shows data-parallelism for an image

Since each work item executes the same code but loads different data, this is data–parallelism. Figure 2.4 shows how work items are organized in *work groups* with a unique ID. For example, consider an image with resolution 512×512, both GLOBALSIZE(0) and GLOBALSIZE(1) are equal to 512. Since there is one kernel execution per pixel, there will be the total of 262144 kernel executions. Work item for pixel $x = 15$, $y = 40$ would have a global ID $(15, 40)$. OpenCL maps all the work items to be executed onto an n–dimensional grid, also called `NDRange` (or an index space). Work groups are executed together[17] and work items within it share resources and can communicate (i.e. streaming multiprocessor)[5]. However, the size of work groups are hardware specific.

2.4.3 Field programmable gate array

Previous decade yielded advances in object–oriented programming for code reuse and parallel computing which resulted in new programming languages, frameworks and tools which allow software engineer to quickly prototype and test different approaches in solving a particular problem. With increasing focus on parallelization and concurrency, field programmable gate arrays (FPGAs) gained attention of the software community. With recent advancement in its programming model and decrease in circuit fabrication cost, FPGAs today enabled the creation of a custom circuit which implements an algorithm using a development process *almost similar* to traditional CPU program-

[17]Work-groups which are executed together in a lockstep are also called *warps* by Nvidia literature, and *wavefronts* by AMD.

ming [122].

FPGA is a type of an integrated circuit (IC), an *empty* chip composed of off–the–shelf basic programmable logic elements called logic cells. Modern FPGAs consist of up to two million logic cells which can be configured to implement a variety of software algorithms. Although traditional FPGA design flow is more similar to a regular IC than a processor, an FPGA provides a significant cost advantages in comparison to an IC development effort and much more flexibility since it is reconfigurable after fabrication. The two main players in the FPGA market today the provide tools to make the FPGA design process much faster; Xilinxs' Vivado High-Level Synthesis (HLS) or Alteras' SDK for OpenCL. Figure 2.5 shows the time necessary to develop an initial version of the same software application and the optimized version for different platforms. As seen, FPGA optimized version provides the best performance, however in the time which exceeds typical design time limit in a software project. Therefore, historically, FPGAs are used only in software projects where the performance was of utmost importance, e.g. aerospace and defense, automotive, medical electronics, ASIC prototyping, high performance computing, scientific instruments, etc. However, with a recent paradigm shift in processor design towards multicore processors and program parallelization, a software engineers need to structure algorithms in a way which leads to efficient parallelization and performance. Therefore FPGA manufactures saw this as an opportunity to increase FPGA presence in software projects. As shown in Figure 2.5 with appearance of new programming models, closer to traditional software design rather to IC design, FPGA programs can be developed faster and time to achieve the optimized version is reduced under the typical design time limit.

For many years, FPGA development was closer to electrical engineering rather than software engineering. The development of new tools, programming model for FPGAs became more similar to CPU programming and offers constructs found in any of the high level languages, such as iterations, selections, functions, etc. However, despite bringing the development process closer to CPU programmers, one still needs a lot of knowledge to understand how FPGAs work in order to gain the best performance. The basic structure of the FPGA consists of: a) *configurable logic blocks* – CLBs, c) *wires* and d) *input/output blocks* as it is shown in Figure 2.6.

Although the structure depicted in Figure 2.6 is sufficient to implement any algorithm, the efficiency of the resulting implementation is limited in terms of computational throughput, resources and clock frequency. Therefore, modern FPGAs also incorporate basic (commonly used) elements such as: embedded memory, phase–locked loops for driving FPGA fabric at different clock rates, high speed serial transceivers, off–chip memory controllers, multiply–accumulate blocks, etc.

Configurable logic blocks consists of a:

– *LUT*, a look up table is the basic building block capable of implementing any logic function of n Boolean variables.

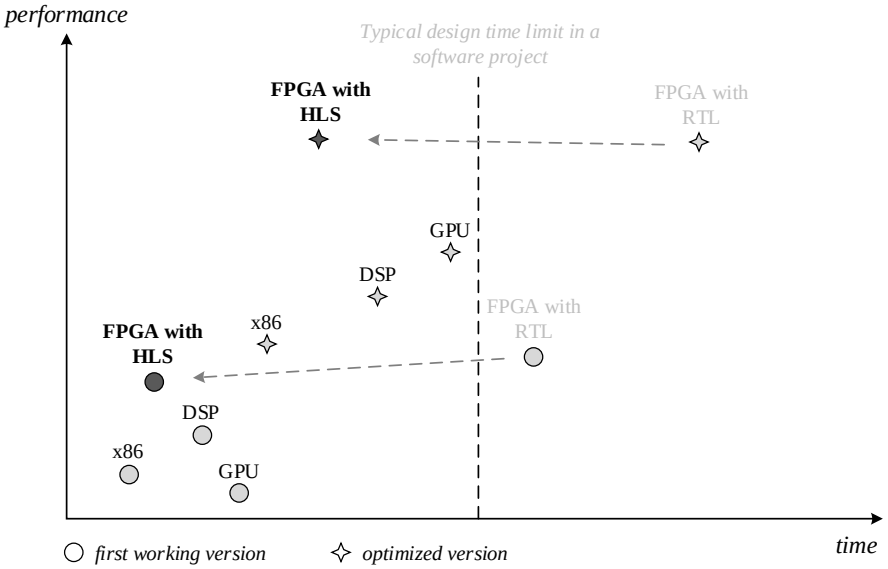

Figure 2.5: Design time vs. application performance (from [122]). A design shift from register–transfel level (RTL) model to hardware–level synthesis (HLS) dramatically reduced the design time while keeping the application performance.

– *Flip–flop*, is the basic storage unit.
– *multiplexor*, is a selection circuit which selects any of the N input lines and feeds it to the appropriate output.

Wires (interconnect) is a set of flexible routing connections which route the signals between CLBs and input/output nodes. There are several different routing types, from the ones designed to interconnect between CLBs, the fast horizontal and vertical long lines spanning across the device, to global low–skew routing for clocking and other global signals. The design software makes the interconnect routing task hidden to the user unless it is specified otherwise, thus significantly reducing design complexity.

Input-output nodes (IOBs) provide the support for dozens of I/O standards and providing an interface between the FPGA and the rest of the platform.

Difference with CPU

The main difference between CPU and FPGA processing is the fixed architecture of CPU. With a processor, the computation architecture is fixed so the job of the compiler is to determine how to best fit the software application in the available processing structures. Performance depends on how well an application maps to the capabilities of a processor. However with FPGAs, compilers job is to create a processing architecture using the available hardware building block which best fit the software program.

Consider a simple function $z = a + b$. FPGA compiler needs to compile the high level

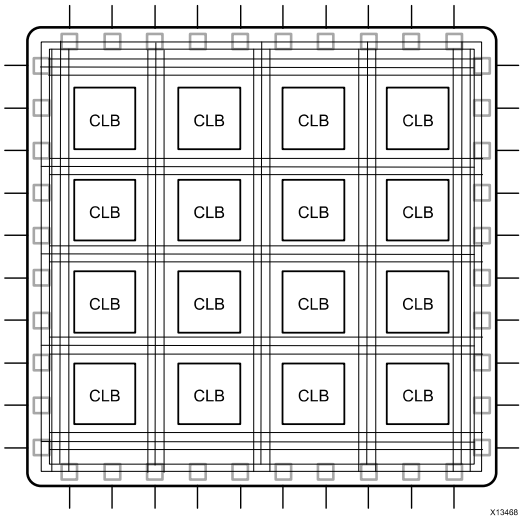

Figure 2.6: FPGA Architecture (from [122])

language into several LUTs required to achieve the size of the output operand[18]. Unlike a processor where all computations share the same ALU, in a FPGA implementation of the previous function the computation of z has a unique dedicated LUT(s). In addition to assigning unique LUT resources per computation, FPGAs also differ from processors in both memory architecture and cost of memory access. In general, the memory access is instantaneous.

The processor clock frequency is one of the parameters which differ greatly between CPU and FPGA. A typical CPU clock frequency is around 2GHz while a high end FPGA would have around 500MHz. Given a choice between CPU and FPGA to develop a software solution, one would easily choose CPU, however the clock rate for these two options is highly misleading. Regardless of a CPU type, the instruction execution always follows the same (simplified) stages: 1) fetch the instruction (IF), 2) decode the instruction (ID), 3) execute (EXE), 4) fetch data for the next instruction (MEM), 5) write back the solution in local registers (or global memory) (WB).

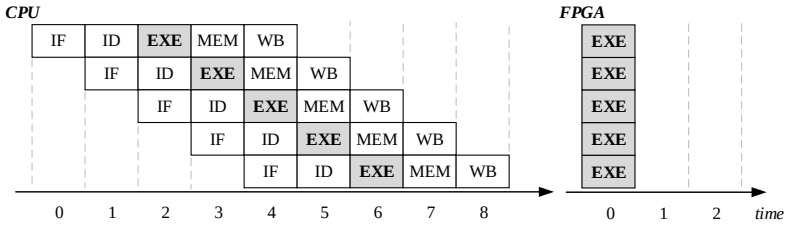

Figure 2.7: Instruction execution, CPU vs FPGA (from [122])

As shown in the left side of Figure 2.7 modern processor are capable of running the instructions with some degree of overlap. This case is for uniprocessor, but the principle

[18] As a general rule, 1LUT is equivalent to 1bit of computation[122]

is the same for multicores with multiplication of the same process. Also, the assumption in the figure is that each of the instructions is executed in one processing cycle. The right side of the Figure 2.7 shows the same execution process for an FPGA. As seen it executes 5 different instructions in parallel. Each EXE is a custom circuit, and any changes in the user application invoke changes to the circuit within FPGA. The full time necessary for application execution in CPU is 9 time units, and in FPGA it's 1.

Latency is the number of clock cycles it takes to complete an instruction, and in the previous example for CPU it equals to 5 clock cycles per instruction. Application latency in both FPGAs and CPUs is usually resolved by applying *pipelining*. For a CPU this means overlapping the instructions which allows for a significant improvement in performance. In the previous example instead latency of 25 (5 per instruction and there were 5 instructions), the total latency was 9.

Consider the example shown in Figure 2.8. The function to be executed on an FPGA consists of 5 building blocks. The time necessary for execution of each block is 2ns, so the total execution time is 10ns. If the block is executed in one cycle, the design is limited to 100MHz frequency (and latency remains 1 cycle). However, if the blocks are organized in a pipeline, i.e. filter like nature, where the source of the next block is the sink of the previous block, the maximum clock frequency is now 500MHz. In contrast to a CPU and a GPU design, the design of an FPGA allows for high flexibility of operation allocation and clock usage. Depending of the usage scenarios, the clock can be increased to reduce latency or decreased to reduce overall power usage[122]. The operations can be multiplied several hundred (or thousand times) across CLBs to achieve extreme parallelism but also increase the throughput (number of cycles necessary to accept the next input data).

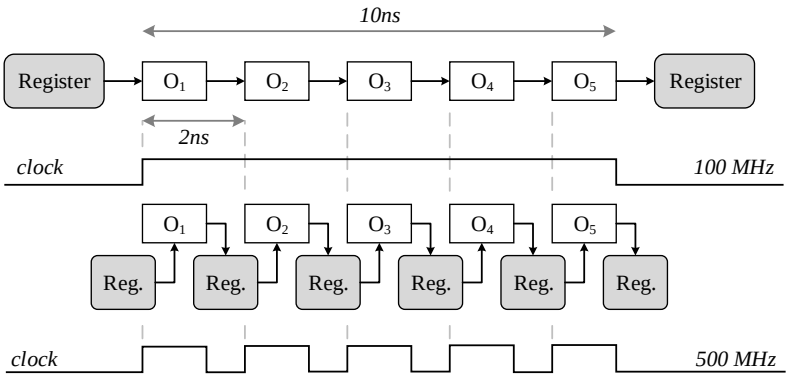

Figure 2.8: FPGA pipelining, depending on usage scenarios, a FPGA developer can choose different implementation options, where as for a CPU or a GPU the design is permanent.

IP cores

FPGA programming is more often done by electronics engineers rather than software engineers. In this domain it is common to create an IC design and to reuse it many times, therefore the similar thinking is applied to FPGA programming, each functionality, once it is programmed a developer can choose to publish it and make it available for other users. Such units of code which are widely available for reuse are called intellectual property cores, IPs. IP is a fundamental building block of FPGA program design. It is the product of human intellect which must be unique, novel and unobvious. IP cores are a blueprint for electronic circuits which enable us to achieve some functionality faster by reusing good, proven and working solutions to a particular known problem. For instance this can be an algorithm for image filtering, MPEG engine, MP3 engine, Ethernet network processor, specific CPU, etc. IP cores are grouped into three categories, soft cores, firm cores and hard cores. Hard IP core is a physical manifestation of the IP design, i.e. a mask layout of a circuit which cannot be changed. Soft cores are synthesizable from HDL and provide better flexibility for the price of predictability. Firm IP cores are in between two previous concepts.

2.5 Summary

The first part of this chapter presents the bond between the computing units and the clarification for the need of multiple computing units of different types. Although it has been shown by previous research that for certain problems GPUs perform a lot better than CPUs, this does not mean that a CPU will be replaced from the current computing models. Rather than replacing it, a GPU augments the computing system with additional capabilities. Similarly, as the FPGA does along with other processing units for accelerating specific casks, e.g. ASIC, ASIP, DSP, etc. However, in the presence of multiple computing units capable of accelerating the same computing task, who decides where does the task get to be executed?
Swebok 3.0 has shown that software engineers need to expand their knowledge areas more towards the hardware. In the latest release it included *architecture decisions* as well as *computing*, *mathematical* and *engineering foundations* as the required knowledge of any software engineer. Other researchers have also shown that hardware–software codesign techniques should be known to anyone who wants to keep up with challenges of increasingly complex electronic systems, including SoC designers, software and hardware engineers.

To cope with increasing complexity of computer systems, software engineers suggest using component based software engineering (CBSE) approach. CBSE has been known in the past as a method which provides a solid foundation for dealing with software engineering challenges related to increased system complexity. It efficiently man-

ages different computing unit types, data structures, processing rates and programming paradigms by separating concerns into smaller manageable units, i.e. software components. To effectively achieve the research objectives laid out by this work, CBSE needs to be applied with considerations to related work in the field of a) software modeling for embedded systems, b) software architecture for embedded systems, c) modeling of heterogeneous systems, d) computing environments which include CPUs, GPUs and FPGAs and e) deal with software component allocation (placing, mapping).

After briefly presenting the actual topics of the related fields, the second part of this chapter provides and overview of heterogeneous computing necessary to follow the course of further research presented in this thesis. This includes an overview of CPUs, CPU families, multicore CPUs and their processing capabilities. Likewise it introduces the GPUs and the processing model which they follow. Finally, this chapter presents FPGAs along with their processing model and all necessary concepts a software engineer should know to understand the subsequent chapters.

CHAPTER 3

MATHEMATICAL MODEL OF A HETEROGENEOUS COMPUTING PLATFORM

To precisely capture the properties of a heterogeneous computing system, software engineering researchers often resort to formal modeling. This chapter introduces the mathematical model for the formal description of a set of heterogeneous computing units, a set of software components, a set of constraints and finally a cost function. The cost function provides a software architect with the most suitable allocation of software components to a heterogeneous computing units with regard to the given constraints, simplifying the multi–criteria architectural decision making process.

3.1 Introduction

The first goal of this research (**RG-1**), is to develop a model which formally describes a set of software components[1], a set of heterogeneous computing units and the relationship which exists (or will exist) among them. The relationship is such that software components perform operations, exhibit certain properties, and are realized on a particular computing unit. The computing unit provides a resources necessary for the realization of a software component.

The realization of a software component for a particular computing unit can also be referred as an allocation; i.e. *a particular software component is allocated on a particular computing unit and it performs functions bound by a set of extra–functional properties (e.g. timing, security, scalability, etc)*. However, depending on the properties of a computing unit, a software component can have different performance, while providing the same functionality. This is especially obvious in heterogeneous systems where computing units are of different types and consist of vastly different computing paradigms.

Allocating the same software component on different computing units results in different component behavior. Since a component oriented software architecture can

[1]Throughout this work, the word component refers to a software component, unless specified otherwise.

have a variety of allocations on a heterogeneous computing system, different allocations can perform very differently, resulting with different behavior, on a system level. Not from the functional, but rather from an extra–functional point of view. Considering this, in a system with multiple software components and computing units, one can wonder, how to allocate software components to computing units; in order to have the best execution time, the maximal memory utilization, or to minimize the energy consumption because the system is battery powered, etc.

These are the questions a software architect faces in an early design phase, and he/she makes decisions which later on define the whole system. In order to answer such questions, a software architect designing such system needs to have access to information about the properties of software components and computing units. While this thesis will not make any assumptions or strict procedures on how to obtain that information, section 4.3 provides some suggestions.

This chapter[2] deals with formalizing all the necessary information for architectural decision making and providing a complete model of a heterogeneous computing platform, resulting with a framework which provides an answer, to which allocation is the best. The first and foremost task is to define what is an allocation.

3.2 Allocation function

The heterogeneous platform consists of two main elements, a set of heterogeneous computing units and a set of software components. In order to realize some functionality, a component must be placed in an execution environment, i.e. a computing unit. This relationship is shown in Figure 3.1. Software components communicate by calling in different operations from other computing units, and this communication, i.e. data flow is realized through physical connections which exist between computing units.
The heterogeneous platform $\mathbb{H}$ consists of the set of software components $c_i \in \mathcal{C}, i = 0, ..., n$ and the set of computing units $u_i \in \mathcal{U}, i = 0, ..., m$, i.e. $\mathbb{H} = (\mathcal{C}, \mathcal{U})$.

An allocation is a mapping of n software components from the set $\mathcal{C}$ to a (sub–) set of computing units $\mathcal{U}'$, with $|\mathcal{P}| = m^n$ being a set containing all possible allocations with the size. The allocation is given by the following function:

Definition 3.1. Function $\alpha : \mathcal{C} \rightarrow \mathcal{U}$ is a component allocation function where $\alpha^{(a)} = (p_1, ..., p_n) \in \mathcal{P}$ defines a particular a–th allocation of components from $\mathcal{C}$ to computing units from $\mathcal{U}$.

According to this definition α is the allocation function, while $\alpha^{(a)}$ is a particular solution vector, i.e. allocation. I is noteworthy to understand that not all the computing units need to necessarily host a component, but all software components need to be allocated,

[2]Some excerpts of this chapter have been published [108], and here they are revised and extensively extended.

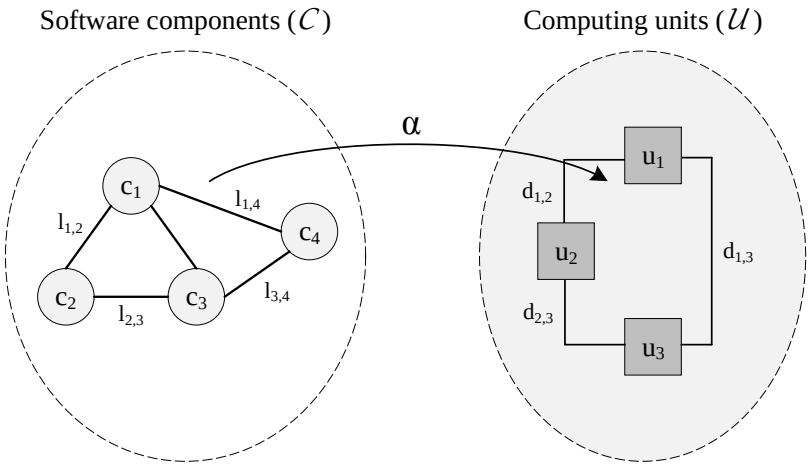

Figure 3.1: Mapping, i.e. allocating software components to computing units

therefore some computing units from $\mathcal{U}$ may remain unused. This can also be written as $\mathcal{U}'$, where $\mathcal{U}' \subseteq \mathcal{U}$.

The following example shows a sample allocation. In this particular case the number of components is equal to number of computing units ($n = m$), and every computing unit hosts one component. The *position* in solution vector $\alpha^{(a)}$ represents a component and its *value* represents the computing unit on witch it is allocated. In this example, software component c_1 is allocated to computing unit u_1, c_2 to u_2, etc.

$$\begin{array}{rcccc} & u_1, & u_2, & \dots, & u_m \\ & \uparrow & \uparrow & & \uparrow \\ \alpha^{(a)} = & (c_1, & c_2, & \dots, & c_n) \end{array}$$

In the real world heterogeneous systems the number of components is very often much larger than the number of computing units, in the solution vector $\alpha^{(a)}$, this would be visible as multiple occurrences of the same value throughout the elements of the vector.

The allocation function α can now successfully generate different allocations. Consider for a while, what allocations really are; different permutations (with repetition) of software components and computing units. To evaluate which allocation is the best, one needs to search through the space of m^n possible answers, where m is the number of computing units ($|\mathcal{U}| = m$), and n is the number of components ($|\mathcal{C}| = n$).
Obviously, the search space increases rapidly with the number of the components and computing units, so to find the optimal component allocation with a respect to a particular goal is (at least) very time–consuming. This is because resource allocation problems are inherently NP–hard [53]. So to find the best allocation it means to find a sub–optimal, good enough solution obtained in a reasonable amount of time. Therefore, one of the challenges of this thesis is to create a cost function which evaluates different allocations and construct it in a way which is easy to automate.

3.3 The allocation cost function

This section presents the construction of the cost function and the component allocation model. The cost function uses the information contained in the model of the heterogeneous platform in order to evaluate any particular allocation. The allocation with the minimal cost is also the best one. The term *best* takes into the consideration the following information defined by a software architect:

a) *resource consumption*, every software component consumes certain amount resources of the computing unit to which it is allocated to, in order to realize its function,
b) *resource availability*, every computing unit has a limited number of resources, and also resources of different types which consequently reflects on the performance of the components hosted by a particular unit,
c) *architectural preference*, any additional logical statement, which for example states that two specific components must be allocated together, while the third one should not be on the same computing unit. This can be achieved by the following constraints:
   i) components that must always be allocated together on the same platform
   ii) components that must not be allocated on the same platform
   iii) computing units must or cannot host same specific components

The component allocation model constructed in this chapter is referred as $\mathbb{M}_\alpha$, while the cost function will be referred to as w. Both $\mathbb{M}_\alpha$ and w will be incrementally built and explained in detail through the subsequent sections[3].

3.3.1 Elements of the cost function

For the component to execute in accordance to its specification, certain resources need to be available. In different systems the relevance of a certain resource changes, in some, one resource might be very important while others can be irrelevant. The choice about which resources to include or exclude from the system design decision process is made by a software architect. While the software architect can specify an arbitrary number of resources, generally, the $\mathbb{M}_\alpha$ model recognizes the following:

a) *processing resources*, resources provided by a computing unit necessary for realizing the functionality of software components
b) *communication resources*, resources provided by a computing unit necessary for realizing data transfer between software components

[3]The reason behind this naming scheme is that letter M suggests the word model, and α is the allocation function. Therefore it could be read as *allocation model*

Processing resource requirements

Having in mind a heterogeneous platform, each component that can be allocated on a different computing unit also uses a different amount of resources depending on the given computing unit type. The specification of the resources required by the components allocated on the computing units is 3–dimensional. The first dimension specifies the components, the second specifies computing units and the third specifies the amount of necessary resources. This is formally written as an array:

Definition 3.2. For n software components, m computing units, and l different resources, $\mathcal{T} = [t_{ijk}]_{(n \times m \times l)}$ is a resource requirement 3-d array where t_{ijk} is the value of the k–th resource required by i–th software component allocated on the j–th computing unit.

With the resource requirement array $\mathcal{T}$, a software architect has the means to formally specify the amount of any resource required by any component while it's allocated on a particular computing unit. For any particular allocation $\alpha^{(a)}$, a software architect can look up the required resources in the array $\mathcal{T}$, add them up together and get the resource cost. Formally, this is formally written as *res* function:

$$\text{res}\left(\alpha^{(a)}\right) = \sum_{k=1}^{l} \sum_{i=1}^{n} t_{ip_ik}, \tag{3.1}$$

where:

t – is the element of the resource consumption array $\mathcal{T}$,
l – is the number of different resources presented in array $\mathcal{T}$,
n – is the number of components,
a – is the a–th allocation vector,
p_i – is the i–th element of $\alpha^{(a)}$.

Communication requirements

While the function res (Equation 3.1) provides an information about the resource consumption of any given allocation, one also needs to account for the communication relationships between components. Communication can be viewed from two different aspects:

a) *software component communication*, some components are used more frequently than others and exchange a large amount of data (their activity is more intensive),
b) *computing unit (hardware) communication*, is bound by physical communication channels and governed by different communication protocols (e.g. LAN, Bluetooth, CAN bus, etc.)

Both hardware and software communication are realized on two different levels, first is the *internal* and second is the *external* communication. The internal communication refers to all communication channels available within a single component or computing unit, while the external communication refers to all the communication channels which exist and bind components or computing units together.
To quantitatively represent the communication between software components, a new matrix is introduced in to the model $\mathbb{M}_\alpha$. It defines the communication *intensity* (e.g. average number of function calls) of each component.

Definition 3.3. $\mathcal{K} = [k_{ij}]_{(n\times n)}$ is a communication intensity matrix where k_{ij} represents a communication intensity approximation between i–th and j–th software component.

If components i and j are not communicating then $k_{ij} = 0$. Also notice that $\mathcal{K}$ is symmetric so the direction of the communication is irrelevant at this point. Additionally, the definition 3.3 does not limit the user on how to quantify the values in matrix $\mathcal{K}$. This matrix is the first of two approximations used by the component allocation model $\mathbb{M}_\alpha$, and as such it should be obtained by an experts' suggestion or approximated with guidance by measurements conducted on a real–world system.

To describe the communication further, one must also take into consideration that different allocations have no impact on the communication intensity. In essence, this means that wherever you place the components, they will always communicate with the same intensity. However, from the perspective of hardware communication, this is not the case. Different allocations make a big difference, since the physical communication paths used by software components may change. And because some of them may be laggy while others my be quite fast, communication channels can be characterized by a *communication cost*. The channels through which the data flows slower are more expensive to use, while other channels with faster data transfer rates are less expensive. This directly reflects on the overall system performance. Components which communicate intensively between computing units with high communication cost have a larger impact on overall performance than those communicating sporadically with less data exchange. In heterogeneous computing environments connected via different types of communication channels (e.g. Ethernet, CAN-bus, Wi-Fi, etc.) this is very common [94]. With that in mind, a new matrix is introduced in to the $\mathbb{M}_\alpha$ model, and it contains information about the communication channel cost between computing units:

Definition 3.4. $\mathcal{C} = [c_{ij}]_{(m\times m)}$ is a platform communication cost matrix where c_{ij} represents a communication cost between i–th and j–th computing unit. For $i = j$, $c_{ij} = 0$.

In addition to the res function which quantifies the performance of any given allocation from the standpoint of processing resources, the new information available in matrices $\mathcal{K}$ and $\mathcal{C}$ can be used to create an additional function which quantifies the communication performance. To find out the communication performance, one needs to look

up the communication intensity between two components (k_{ij}) and multiply it by the communication cost of the communication channel (c_{ij}). Formally, this is written as:

$$\text{com}\left(\alpha^{(a)}\right) = \sum_{i \leq j} k_{ij} \cdot c_{p_i p_j}, \tag{3.2}$$

where,

k – is the element of the communication intensity matrix $\mathcal{K}$,
c – is the element of the platform communication cost matrix $\mathcal{C}$.

Finally, to get the cost of any allocation, equations Equation 3.1 and Equation 3.2 are joined in to a single cost function $w\left(\alpha^{(a)}\right)$:

$$w\left(\alpha^{(a)}\right) = \text{res}\left(\alpha^{(a)}\right) + \text{com}\left(\alpha^{(a)}\right) \tag{3.3}$$

When expanded, the final form of the cost function is:

$$w\left(\alpha^{(a)}\right) = \sum_{k=1}^{l} \sum_{i=1}^{n} t_{ip_ik} + \sum_{i \leq j} k_{ij} \cdot c_{p_i p_j} \tag{3.4}$$

Although the current form of the cost function may be used for some purposes, it is noteworthy to understand that there are some issues which have not been addressed. The major issue here, presents the freedom given to a software architect to define any number of different resources of different types. Consequently for the cost function w this means that it deals with different measurement units with different orders of magnitude. For example, having average memory usage in *megabytes* and average latency in *milliseconds*, one does not simply compare these units, especially in formal expressions. This presents a big issue since *apples and oranges* cannot be compared. Further discussion about this issue and ways of handling it are dealt with in section 3.5. Before this, the model $\mathbb{M}_\alpha$ will be introduced with constraints presented in the following section.

3.4 Constraints

The Equation 3.19 accounts for both processing and communication resource cost for any given allocation. As such, it provides a good starting point for further extensions,

namely including constraints. Since there are some allocations that are infeasible to implement, the model $\mathbb{M}_\alpha$ needs to be extended to contain such information. It should consider:

a) *limitation of available resources* (both processing and communication resources), each computing platform has limited resources, and some allocations can require more resources than a computing system can provide,
b) *architectural preference*, a software architect can specify additional constraints, such as which two components must or must not be allocated on the same computing unit, etc.

If an allocation requires more resources than it is available, or if it does not submit to constraints defined by an architect, it needs to be dismissed. To include these constraints into the model $\mathbb{M}_\alpha$, it will be expanded with new information for dismissing infeasible allocations.

3.4.1 Dismissing infeasible allocations due to limited resources

The current component allocation model consisting of matrices $\mathcal{T}, \mathcal{C}, \mathcal{K}$, does not provide the information about a resource availability. For that purpose, $\mathbb{M}_\alpha$ is further extended with two new matrices. The first one contains the information about the maximal availability of processing resource provided by a computing system. It is defined as:

Definition 3.5. $\mathcal{R} = [r_{jk}]_{(m \times l)}$ is a computing unit resource matrix where r_{jk} represents k–th resource of a j–th computing unit.

The second matrix contains the information about physical communication constraint, i.e. the bandwidth available between computing units. It is defined as follows:

Definition 3.6. $\mathcal{B} = [b_{ij}]_{(m \times m)}$ is a bandwith matrix where b_{ij} represents communication bandwidth available between i–th and j–th computing unit.

With this additional information contained in matrices $\mathcal{R}$ and $\mathcal{B}$, a software architect can account for the resource limitation and dismiss invalid allocations. For the verification of invalid allocations, it is necessary to construct new parameters for the cost function w. In cases where the given allocation is invalid, these parameters would render the result of the cost function useless. An intuitive way to do so is to multiply the cost function with a parameter equal to 0 if the allocation is invalid. Therefore any allocation for which $w\left(\alpha^{(a)}\right) = 0$, is declared as invalid and subsequently dismissed.

For this purpose, the cost function is updated with two factors. These factors are functions which verify the availability of resources for a particular allocation. The first one is resource constraint function ρ, and the second one is communication constraint function κ. Resource constraint function ρ is defined as follows:

$$\rho\left(\alpha^{(a)}\right) = \begin{cases} 1, & \text{if } a\text{–th allocation does not exceed the maximum available processing} \\ & \text{resources} \\ 0, & \text{otherwise} \end{cases}$$

Formally, it this is written as:

$$\rho\left(\alpha^{(a)}\right) = \begin{cases} 1, \text{if} \sum_{i=1}^{n} \sum_{k=1}^{l} \left(t_{ip_ik}\right) < \sum_{j=1}^{l} r_{p_ij} \\ 0, \text{otherwise} \end{cases} \tag{3.5}$$

where r – is the element of computing unit resource matrix $\mathcal{R}$.

When the function ρ is added as a factor in the resource cost function defined by Equation 3.1, it then becomes:

$$\text{res}\left(\alpha^{(a)}\right) = \left(\sum_{k=1}^{l} \sum_{i=1}^{n} t_{ip_ik}\right) \cdot \rho\left(\alpha^{(a)}\right) \tag{3.6}$$

The resource cost function (Equation 3.6) now disregards any solution which exceeds the available processing resources. Similarly to this, the communication cost function is updated with function κ as a factor which invalidates the allocation if it exceeds the communication cost. The new function can also result in values 1 or 0 depending on the following:

$$\kappa\left(\alpha^{(a)}\right) = \begin{cases} 1, & \text{if } a\text{–th allocation does not exceed the maximum available bandwith} \\ 0, & \text{otherwise} \end{cases}$$

Formally written, the expression for κ becomes:

$$\kappa\left(\alpha^{(a)}\right) = \begin{cases} 1, \text{if} \sum_{i \leq j} \left(k_{ij} \cdot c_{p_ip_j}\right) < \sum_{i \leq j} b_{p_ij} \\ 0, \text{otherwise} \end{cases} \tag{3.7}$$

With the communication constraint defined, the Equation 3.2 can be updated to:

$$\text{com}\left(\alpha^{(a)}\right) = \left(\sum_{i \leq j} k_{ij} \cdot c_{p_ip_j}\right) \cdot \kappa\left(\alpha^{(a)}\right) \tag{3.8}$$

To simplify the notation of the individual constraint functions (i.e. factors) are grouped in a single constraint function ctr:

$$\text{ctr}\left(\alpha^{(a)}\right) = \rho\left(\alpha^{(a)}\right) \cdot \kappa\left(\alpha^{(a)}\right) \tag{3.9}$$

Having defined the constraints for processing and communication resources, the previous version of the allocation cost function can be updated to:

$$w\left(\alpha^{(a)}\right) = \left(\text{res}\left(\alpha^{(a)}\right) + \text{com}\left(\alpha^{(a)}\right)\right) \cdot \text{ctr}\left(\alpha^{(a)}\right) \tag{3.10}$$

when expanded, its complete form of the allocation cost function is:

$$w\left(\alpha^{(a)}\right) = \left(\sum_{k=1}^{l}\sum_{i=1}^{n} t_{ip_ik} + \sum_{i\leq j} k_{ij} \cdot c_{p_ip_j}\right) \cdot \text{ctr}\left(\alpha^{(a)}\right) \tag{3.11}$$

The function ctr will also be referred as *solution validity product*. If the product of all its element is 1, the solution is valid. With this, the allocation cost function can account for any constraint that comes out of physical and real world limitations. However, as previously said, a software architect can define additional limitations which also affect the final allocation. This is further described in the following section.

3.4.2 Dismissing infeasible allocations due to architectural specification

The current, fairly broad, model of the component allocation can incorporate information about all aspects of the system. It contains information about availability and demand of processing resources and communication resources, which is enough to generate feasible allocations. However, in the real world, the allocation decision can very rarely be made purely on hardware information alone. Often, there are human made requirements which come into place because of several factors:

a) *previous experience*, although a certain allocation can be declared the best one purely based on numbers, there could be some previous experience which suggest otherwise. Mainly because of the factors which are hard to represent in the model, for example; bad software drivers for certain hardware components, poor documentation of components or computing units, incompatible systems, etc.
b) *reusability*, there might be legacy components available from previous successful projects. If they work without issues, a software architect would gladly reuse previously proven and robust solutions, rather than develop new ones. This is a

typical architectural sacrifice of performance in exchange for higher reliability.

c) *development effort*, in some cases the optimal allocation could be the one which is hard to implement and it would take more development effort than other allocation which would perform slightly poorer, but would take lot less effort to implement (e.g. an FPGA vs a CPU).

d) *price*, the best performing allocation can possibly also be the most expensive one, therefore a software architect could sacrifice performance for a lower price.

To incorporate such architectural knowledge into the allocation cost function, in this section the model $\mathbb{M}_\alpha$ will be further extended. The extension will allow the specification of the following constraints:

a) *hosting capability*, which provides an option to point out which components can or can not be allocated to certain computing units.

b) *mandatory joint allocations*, which provides an option to point out which components must mandatory be allocated together on the same computing unit.

c) *forbidden joint allocations*, which provides an option to point out which components must not be allocated together on the same computing unit.

The constraint function ctr $\left(\alpha^{(a)}\right)$ will be updated with these new constraints. Similarly as for previously defined constraints, the new ones will invalidate the entire allocation if these new constraints, i.e. conditions aren't met.

Component hosting capability

The first architectural constraint that is introduced to the constraint function is the computing unit *hosting capability matrix* $\mathcal{D}$:

Definition 3.7. $\mathcal{D} = [d_{ij}]_{(n\times m)}$ is the computing unit hosting capability matrix, where d_{ij} represents a statement whether the i–th software component can be hosted on the j–th computing unit.

The value of d_{ij} is either 0 or 1, depending on the following condition:

$$d_{ij} = \begin{cases} 1, & \text{if } i\text{–th software component cannot be allocated to } j\text{–th computing unit} \\ 0, & \text{otherwise} \end{cases}$$

To verify the architectural constraint from the previous definition (3.7), it is necessary to construct a mathematical function which accordingly verifies the validity of the allocation. This means that each pair of elements in an allocation $\alpha^{(a)} = (p_1, ..., p_n)$, i.e. solution vector needs to be evaluated against the matrix $\mathcal{D}$. If any evaluation results in 0, the solution is not valid. Essentially, the constraint function needs to count all the zeros of some allocation in accordance with matrix $\mathcal{D}$. This function is defined as:

$$\delta\left(\alpha^{(a)}\right) = \begin{cases} 1, \text{if } \sum_{i=1}^{n}\left(d_{ip_i}\right) = 0 \\ 0, \text{otherwise} \end{cases} \tag{3.12}$$

Mandatory joint allocations

As mentioned previously, a software architect can choose which components need to be allocated together, regardless of the computing unit. For this purpose, a new matrix is introduced to the model to contain this information. It is defined as:

Definition 3.8. $\mathcal{Y} = [y_{ij}]_{(n\times n)}$ is the matrix of mandatory joint allocations, where y_{ij} represents a statement whether the i–th and j–th software component should be allocated together on the same computing unit.

The values y_{ij} of the matrix $\mathcal{Y}$ can be 0 or 1 depending on the following conditions:

$$y_{ij} = \begin{cases} 1, & \text{if } i\text{–th and } j\text{–th component must be allocated on the same computing unit} \\ 0, & \text{otherwise} \end{cases}$$

To verify this constraint (3.8), similarly as in previous section, it is necessary to construct a new function. This time, the function which will result in 1 if the allocation is valid according to $\mathcal{Y}$, and with 0 otherwise. This means that for each element of the vector $\alpha^{(a)}$, this function needs to use the look up matrix $\mathcal{Y}$ and verify if the component is supposed to be allocated with some other component on the same computing unit. For the solution vector $\alpha^{(a)}$ this means that the same value (recall that the value of the vector represents the computing unit) occurs on multiple (exact) positions (recall that position of the vector represents the component). Essentially, this means counting for invalid occurrences of computing units ($u \in \mathcal{U}$) in the solution vector according to matrix $\mathcal{Y}$. To clarify further, consider the following example. The heterogeneous system consists of 4 software components and 3 computing units, $\mathcal{C} = \{c_1, c_2, c_3, c_4\}$, $\mathcal{U} = \{u_1, u_2, u_3\}$.

$$\mathcal{Y} = \begin{bmatrix} - & 0 & 1 & 0 \\ 0 & - & 0 & 1 \\ 1 & 0 & - & 0 \\ 0 & 1 & 0 & - \end{bmatrix}, \text{and the solution vector is: } \alpha^{(a)} = (u_1, u_2, u_1, u_3)$$

This means that the components c_1, c_3 and the components c_2, c_4 must be allocated on the same computing unit. Here is the truth table which verifies if this is so:

	$\mathcal{Y}$	α	$\neg\mathcal{Y} \wedge \alpha$	
c_1, c_2	0	0	1	
c_1, c_3	1	1	1	
c_1, c_4	0	0	1	
c_2, c_3	0	0	1	
c_2, c_4	1	0	0	$\leftarrow$ bad allocation
c_3, c_4	0	0	1	

Although the above example seems to suggest that the XOR operator would be in order, it fails to demonstrate what happens when a $y_{ij} = 0$ and $p_i = 1$. In such cases XOR would eliminate valid solutions. Therefore, the only case where a solution can be invalid is when $y_{ij} = 1$, in all other cases it is always valid. As seen in this example, the mandatory allocation which states that components c_2 and c_4 need to be allocated together is not satisfied. Formally, this verification can be written as a function in the following expression:

$$\upsilon\left(\alpha^{(a)}\right) = \begin{cases} 1, \sum_{i<j} \neg\left[\neg y_{ij} \wedge [p_i = p_j]\right] = 0 \\ 0, \text{otherwise} \end{cases} \tag{3.13}$$

Note that in the definition, $[\,]$ represents Iverson bracket notation defined as:

$$[P] = \begin{cases} 1, & \text{if } P \text{ is true} \\ 0, & \text{otherwise} \end{cases}$$

Forbidden joint allocations

The final constraint which a software architect can define is the forbidden allocation. It can happen that for reasons previously described, an allocation where two or more components are allocated to the same computing unit is forbidden. The information about these allocations is contained in the matrix $\mathcal{X}$, defined as:

Definition 3.9. $\mathcal{X} = [x_{ij}]_{(n\times n)}$ is the matrix of forbidden joint allocations, where x_{ij} represents a statement whether the i–th and j–th software component must not be allocated together on the same computing unit.

The values in the matrix $\mathcal{X}$ can be 0 or 1 depending on the following cases:

$$x_{ij} = \begin{cases} 1, & \text{if } i\text{–th and } j\text{–th component must be allocated on separate computing units} \\ 0, & \text{otherwise} \end{cases} \tag{3.14}$$

In a very similar fashion as for the mandatory allocations, each element of an allocation $\alpha^{(a)}$ needs to be verified against the matrix $\mathcal{X}$. Consider the following example:

$$\mathcal{X} = \begin{bmatrix} - & 0 & 1 & 0 \\ 0 & - & 0 & 1 \\ 1 & 0 & - & 0 \\ 0 & 1 & 0 & - \end{bmatrix}, \text{and the solution vector is: } \alpha = (u_1, u_2, u_1, u_3)$$

The following truth table evaluates the validity of the solution α:

	$\mathcal{X}$	α	$\wedge$	
c_1, c_2	0	0	0	
c_1, c_3	1	1	1	$\leftarrow$ bad allocation
c_1, c_4	0	0	0	
c_2, c_3	0	0	0	
c_2, c_4	1	0	0	
c_3, c_4	0	0	0	

As seen in the matrix $\mathcal{X}$, the components c_1 and c_3, and the components c_2 and c_4 must not be allocated on the same computing unit. Since the allocation $\alpha^{(a)}$ allocates the components c_1 and c_3 together it is not valid. The logical operator to find out invalid solutions is AND. Therefore, the verification function will be defined as:

$$\chi\left(\alpha^{(a)}\right) = \begin{cases} 1, \text{if } \sum_{i<j} \Big[x_{ij} \wedge [p_i = p_j]\Big] = 0 \\ 0, \text{otherwise} \end{cases} \tag{3.15}$$

Where $[\,]$, as previously, represents Iverson bracket notation. Also, notice that there is a relationship between matrices $\mathcal{Y}$ and $\mathcal{X}$ according to which $y_{ij} \in \mathcal{Y}$ and $x_{ij} \in \mathcal{S}$ follows that $y_{ij} = \neg x_{ij}$ for $i, j = 1, ..., n$ and $y_{ij} = 1$. This means that if two components must be allocated together, it cannot happen that they are also forbidden to be allocated together.

Finally, the constraint function (Equation 3.9) can be updated with the constraint function for verifying allocation preference(Equation 3.12), mandatory allocation (Equa-

tion 3.13) and forbidden allocations (Equation 3.15). The final form of the constraint function becomes:

$$\text{ctr}\left(\alpha^{(a)}\right) = \rho\left(\alpha^{(a)}\right) \cdot \kappa\left(\alpha^{(a)}\right) \cdot \delta\left(\alpha^{(a)}\right) \cdot \upsilon\left(\alpha^{(a)}\right) \cdot \chi\left(\alpha^{(a)}\right) \tag{3.16}$$

With this being the final form of the constraint function, the cost function can now account for both physical resource limitations and limitations defined by a software architect.

3.5 AHP – handling different measurement units

The role of the software architect, among many, is also to decide which resources will be taken in consideration to evaluate different allocations. If an architect chooses to evaluate allocations according to execution time, memory usage and energy consumption, the cost function respectively deals with milliseconds, megabytes and watt–hours. These are three different measurement units which can easily have different orders of magnitude. Therefore, while the previously defined cost function w (Equation 3.11) can compare and rank valid allocations, the problem with different types of resources has still not been addressed. Also, these resources might have different importance. In some scenarios one might prefer that the processing time is twice more important than memory. To solve this problem, Analytic Hierarchy Process (AHP) will be used [96].

AHP is an effective tool used to deal with for complex multidimensional choices, alternatives and tradeoffs. The decision maker is given the option to set priorities to different information upon which the best choice is made. A benefit of applying AHP is that it does not assume that the best choice is the one which consists of optimums of individual choices. With AHP, a decision maker generates a weight for each criterion. The higher the weight, the more important the corresponding criterion is. Different criteria is weighted by using a *pairwise comparison*, by which the decision maker states the importance of each criteria, while the AHP provides a mechanism to evaluate the consistency of these statements. The procedure is mostly written and done in a tree structure, where each branch represents a criterion. For the each depth level, a pairwise comparison needs to take place. The size of the tree can be of arbitrary depth, and each branch will have the different importance in the final decision. More on this topic can be found in [97].

3.5.1 Applying AHP to decision making in component allocation

To apply the Analytic Hierarchy Process to making architectural decisions about allocating software components on a heterogeneous computing system, one level of hier-

archy is sufficient. The decision tree is shown in (Figure 3.2). Each branch represents one criteria which corresponds to one resource. The importance of each resource is weighted using a trade–off vector F. The values it contains are calculated using AHP in tree steps:

1) evaluate the resource importance with pairwise comparison matrix M_c,
2) calculate the principal eigenvector of the M_c,
3) asses the consistency of pairwise comparison.

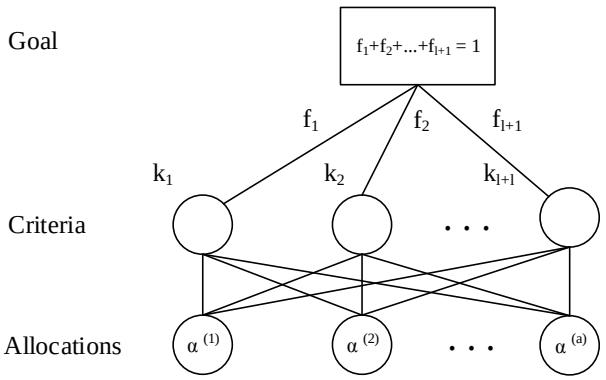

Figure 3.2: Hierarchy for defining the criteria

Pairwise comparison

The first step in AHP is to create a comparison matrix M_c which contains all the resources. The number of resources is defined in the resource consumption matrix $\mathcal{T}$ and it is k. Since the cost function treats communication as a resource on its own, the size of the comparison matrix is M_c is $(k+1) \times (k+1)$, where this one extra element represents the communication. M_c has the following form:

$$\begin{array}{c} \\ k_1 \\ k_2 \\ \vdots \\ k_{l+1} \end{array} \begin{array}{c} \begin{array}{cccc} k_1 & k_2 & \dots & k_{l+1} \end{array} \\ \begin{pmatrix} 1 & m_{1,2} & \dots & m_{1,l+1} \\ (m_{1,2})^{-1} & 1 & \dots & m_{2,l+1} \\ \vdots & \vdots & \ddots & \vdots \\ (m_{1,l+1})^{-1} & (m_{2,l+1})^{-1} & \dots & 1 \end{pmatrix} \end{array} = M_c \tag{3.17}$$

Each resource, i.e. decision criteria needs to be compared with all other elements in the matrix M_c. The elements m of the matrix usually have values which span from $1/9$ to 9. This is the standard AHP scale for comparing criteria and it is interpreted as follows: *1 – equal importance, 3 – slightly favoring, 5 – strong favors, 7 – very strong favoring, 9 – extreme favors.*

For instance, if resource k_1 is slightly more important than resource k_2, consequently $m_{1,2} = 3$, and k_2 compared to k_1 would give the reciprocal value, $m_{2,1} = {}^1/_3$. For $i = j$, $m = 1$.

Calculating the principal eigenvector

The next step of AHP is to calculate normalized principal eigenvector and principal eigenvalue[4]. If A is a square matrix, a non–zeoro vector C is called eigenvector *iff* there exists a number λ, such that $AC = \lambda C$. If indeed λ exists it is called an eigenvalue of matrix A. The vector C is called eigenvector associated to the eigenvalue λ. The largest eigenvalue is called principal eigenvalue (λ_{max}). The eigenvector which corresponds to principal the eigenvalue is called principal eigenvector (ω^*). To use it in AHP, the principal eigenvector is normalized so that the sum of all elements equals 1. These values are elements of the trade–off vector F used to provide weights to different criteria, or in this case to different resources. With the introduction of F to Equation 3.11, the cost function w becomes multi–criterion.

Verifying the consistency of pairwise comparison

The final step, before applying the weight vector F in the cost function is to verify the consistency of a pairwise comparison. Since this is subjected to human judgment it is prone to inconsistency. The improvement of consistent human input attributes to validity of decision priorities. AHP deals with this issue by measuring the consistency of prioritization in matrix M_c. Matrix M_c is consistent if $m_{ij}, m_{jk} = m_{ik}$ for all i, j, k. The consistency is measured by calculating the consistency ratio C_R. It is given as $C_R = {}^{C_I}/_{R_I}$, where C_I is the consistency index and R_I is the random consistency index. C_I is calculated as:

$$C_I = \frac{\lambda_{max} - l}{l - 1} \tag{3.18}$$

where l is the number of different criteria. Since the model $\mathbb{M}_\alpha$ defines k different resources and communication, $l = k + 1$.
Random consistency index R_I can usually be looked up in pre–calculated tables. However, if the tables do not contain the necessary values, one can easily calculate R_I. More on this topic can be found in [97]. Finally, with factor consistency index and random consistency index, one can calculate the consistency ratio. If is $C_R \leq 10\%$, the inconsistency of pairwise comparison is acceptable, otherwise, it's not, and the pairwise comparison should be repeated.

Since different criteria, i.e. heterogeneous platform resources are measured in different units, it is likely that some values may differ in order of magnitude (e.g. tens of

[4]The reason why the principal eigenvector is used is that it invariant under hierarchic composition of its own judgment matrix [98]

milliseconds, thousands of megabytes or hundredths of milliamperes per hour). Hence, before the calculation, the input matrices $\mathcal{T}$, $\mathcal{R}$ and $\mathcal{C}$ need to be normalized, so all the values are in range from 0 to 1. 1 is representing a maximum available amount of a resource. Therefore, F becomes the only factor directly influencing the ratio of importance of a certain resource in the final decision.
The expanded cost function, updated with the trade–off vector F is:

$$w\left(\alpha^{(a)}\right) = \left(\sum_{k=1}^{l} f_k \sum_{i=1}^{n} t_{ip_ik} + f_c \sum_{i \leq j} k_{ij} \cdot c_{p_ip_j}\right) \cdot \text{ctr}\left(\alpha^{(a)}\right) \tag{3.19}$$

Where f_c is the last element of the vector, representing the importance of communication, and f_k are the importance parameters for the non–communication resources. The cost function now accounts for multiple–criteria.

Illustrative AHP example

To illustrate all steps of the AHP, consider a system with four software allocations among which a software architect needs to choose. This choice should be made with consideration to the overall system behavior for each allocation with regard to; system availability (measured as percentage of up–time), systems' average power consumption (measured in watts), average latency of the system (measured in milliseconds) and testability (measured as percentage of code coverage). Applying AHP addresses the following problems; first, it dismisses the issue of different measurement units and second, it results in consistent weights of the selected criteria.

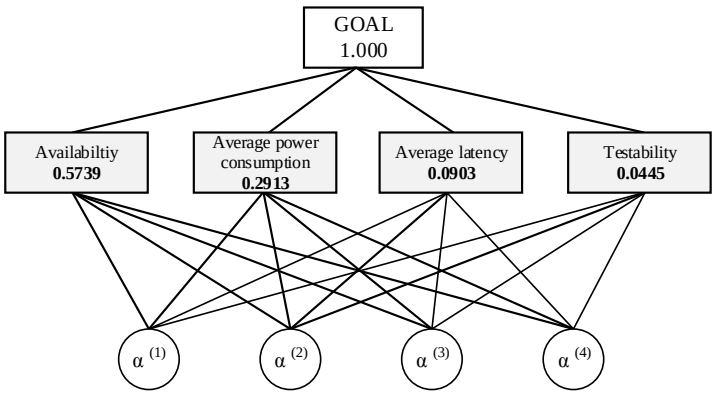

Figure 3.3: AHP hierarchy example with four criteria and four alternatives

The first step of the AHP is to create a decision tree as shown in Figure 3.3. There are four alternatives amongst which one should make a choice based on four criteria, each of which contributing with different weight to the final decision. The weights are

obtained by performing a pairwise resource comparison, i.e. by creating a M_c matrix, as shown in the following expression (3.20):

$$\begin{array}{l} \\ \text{availability} \\ \text{avg.pwr.cons.} \\ \text{latency} \\ \text{testability} \end{array} \begin{array}{c} \begin{array}{cccc} \text{availability} & \text{avg.pwr.cons.} & \text{latency} & \text{testability} \end{array} \\ \begin{pmatrix} 1 & 3 & 7 & 9 \\ 1/3 & 1 & 5 & 7 \\ 1/7 & 1/5 & 1 & 3 \\ 1/9 & 1/7 & 1/3 & 1 \end{pmatrix} \end{array} = M_c \tag{3.20}$$

The availability is the most important criteria, it is 3 times more important than average power consumption, 7 times more than latency and 9 times more than testability. It follows by the average power consumption, latency and testability. By a coincidence their order in the matrix M_c is given by their importance, however this is usually not the case.

The Eigenvector of a given resource comparison matrix results to $\lambda_{max} = 4.2692$. The consistency index is $CI = 0.0897$ which gives the consistency ratio of $CR = 9.97\%$. Since it is smaller than 10%, the comparison of the resources can be considered as consistent. The resulting priority vector is $(0.5739, 0.2913, 0.0903, 0.0445)$, as shown in figure Figure 3.3.

The final step is to consider the input values for each criteria given by the each allocation and evaluate their weight by some function. Table 3.1 illustrates an example with input values for each criteria and allocation, weighting particular criteria and obtaining the best allocation by a simple summing function. According to all the provided information, the best performing allocation should be $\alpha^{(4)}$.

Table 3.1: AHP example resource weigting. Each input is normalized and weighted. The final result is obtained by summation of all parameters, less is better.

	Raw inputs				Normalized				Weighted			
	$\alpha^{(1)}$	$\alpha^{(2)}$	$\alpha^{(3)}$	$\alpha^{(4)}$	$\alpha^{(1)}$	$\alpha^{(2)}$	$\alpha^{(3)}$	$\alpha^{(4)}$	$\alpha^{(1)}$	$\alpha^{(2)}$	$\alpha^{(3)}$	$\alpha^{(4)}$
Availability (unavailable in %)	2	1	5	1	0.2222	0.1111	0.5556	0.1111	0.1275	0.0638	0.3188	0.0638
Average power consumption (W)	10	11	9	10	0.2500	0.2750	0.2250	0.2500	0.0728	0.0801	0.0655	0.0728
Average latency (ms)	130	120	90	100	0.2955	0.2727	0.2045	0.2273	0.0267	0.0246	0.0185	0.0205
Testability (%)	90	40	60	50	0.3750	0.1667	0.2500	0.2083	0.0167	0.0074	0.0111	0.0093
								Final	0.2437	0.1759	0.4140	**0.1664**

AHP can also contain sub–criteria with more complex trees and multiple levels. In such cases, the same procedure is repeated on each sub–criteria level, but then, instead achieving a goal, a sub–criterion should achieve the criterion goal. The total weight of sub–criteria is equal to the weight of the criteria to which it belongs. Weighs from all levels always sum up to 1.

3.6 Accounting for the *synergy effect*

The current cost function, given by the Equation 3.19 evaluates an allocation and attributes it with the number which reflects the weight of the solution with the respect to multiple criteria. It accounts for processing resources, communication resources, physical constraints, architectural constraints and constraints specified by an software architect. Consider for a moment the current form of the *res* function:

$$\text{res}\left(\alpha^{(a)}\right) = \left(\sum_{k=1}^{l} f_k \sum_{i=1}^{n} t_{ip_ik}\right) \cdot \rho\left(\alpha^{(a)}\right) \tag{3.21}$$

Notice that the inner sum iterates and adds up the required resources from $\mathcal{T}$ without any regard to the possibility that some components may already be allocated on a particular unit. This means that any new component allocated to a certain computing unit always takes up the same amount of resources and the performance does not decay. In the real world scenario this will very rarely be the case. Consider a CPU which hosts 5 software components, and upon them another 50 were added. The previous 5, or for that matter all 55 components would hardly perform in the same way, as if there were only one component allocated on a CPU. In some cases, new components might perform better (due to buffering algorithms, sharing resources, memory prediction, etc.), and in other cases it would affect all the components so they would perform poorer. In this work, this will be refereed to as the *synergy effect*, i.e. the effect of stacking up components on the same computing unit. This is one of the essential characteristics of any given allocation and the cost function w should account for this effect. Synergy effect can have positive or negative impact on the overall cost of a certain allocation. The current *res* function does not account for effects which emerge when a single computing unit hosts multiple software components.

Examining the type of a resource, there can be two types of availability of a resource with respect to the synergy effect:

a) *saturable availability*, by adding new components to a computing unit, the performance of all components declines (decays), non the less, new components can still be added,
b) *limiting availability*, with addition of new components to a computing unit, the resources are increasingly consumed until they are fully consumed and new components cannot be added.

Consider the example shown in Figure 3.4. The left side of the figure represents saturated resource availability. With addition of new components, at a certain point a computing unit becomes saturated increasing the resource consumption and performance decay. Also notice that in this example, the performance decay does not grow linearly, meaning that there is a positive synergy effect. However, at a certain point, A comput-

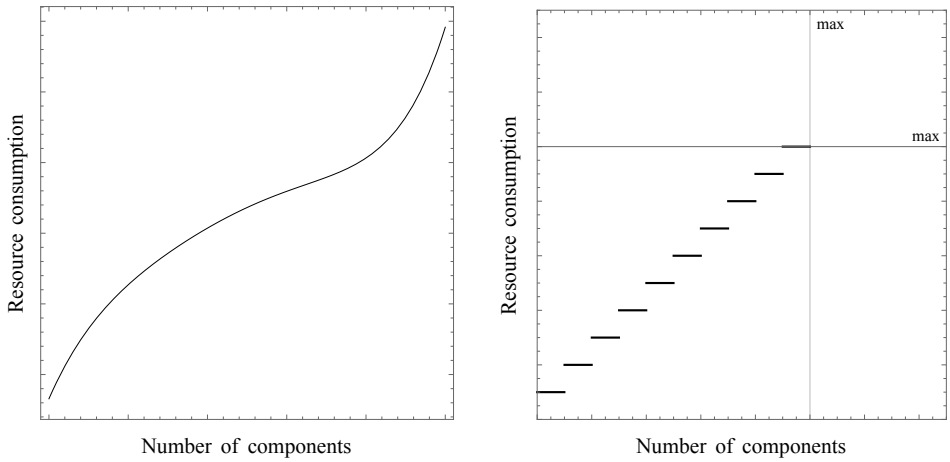

Figure 3.4: Left: saturable availability, right: limiting availability

ing unit is saturated and performance decay becomes to grow exponentially (but new components can still be added). The right side of the figure represents an example of limiting resource availability. Each new component allocated on a computing unit takes up the same amount of resources, growing linearly up to the point where the resources are completely consumed.

Information about synergy effect, i.e. about the performance of software components allocated on computing units with different load, is approximated in the 3-dimensional array $\mathcal{S}$. First dimension represents computing units, second dimension represents the number of components allocated to a computing unit (where maximal size is equal to the total number of components (n)) and the third dimension represents a synergy effect factor for different resources. Array $\mathcal{S}$ is defined as:

Definition 3.10. For n software components, m computing units, and l different resources, $\mathcal{S} = [s_{ijk}]_{(n \times m \times l)}$ is a synergy factor approximation array where s_{ijk} is the factor which compensates for the synergy effect of k–th resource on j–th computing unit while it hosts i software components.

If s_{ijk} is less than 1, it means that new components added to a computing unit uses less resources than a single component allocated on that unit. This is a positive synergy effect. If s is bigger than 1, this is a negative synergy effect and it means that new components consume more resources than one component by itself. To select the right factor from the array $\mathcal{S}$, one would need need to know how many components a specific unit hosts (for a given allocation $\alpha^{(a)}$), and multiply the number of required resources t from the array $\mathcal{T}$ with the proper s. To construct a function for selecting the right factor s from the array $\mathcal{S}$ consider the following example. The heterogeneous system consists of 4 software components and 3 computing units, $\mathcal{C} = \{c_1, c_2, c_3, c_4\}$, $\mathcal{U} = \{u_1, u_2, u_3\}$. Array $\mathcal{S}$ is given as:

$$\mathcal{S} = \begin{bmatrix} 1 & 1 & 1 \\ 0.9 & 0.7 & 1 \\ 0.95 & 0.75 & 1 \\ 1.2 & 1.4 & 1 \end{bmatrix}$$, and the solution vector is: $\alpha = (u_1, u_2, u_1, u_3)$

In this example, $s_{22} = 0.7$ means that if a current allocation has 2 components allocated on the u_2, the resource requirement given by the array $\mathcal{T}$ should be multiplied by 0.7, i.e. there is a positive synergy effect. If the same computing unit hosts 4 components, $s_{24} = 1.4$ and this is a negative synergy effect. Also, notice that computing unit u_3 has no synergy effect since all the factors are 1.

To modify the current *res* function, one final function needs to be constructed. The input to this new function is an allocation along with a computing unit of interest, and it would result in the number of components allocated on the given computing unit. This number is then used to find the right s in the array $\mathcal{S}$. This new function is defined as:

$$\eta\left(\alpha^{(a)}, j\right) = \sum_{i=1}^{n} \left[p_j = p_i\right] \tag{3.22}$$

Where $[\,]$ is the Iverson bracket notation. If you consider the solution vector $\alpha^{(a)}$ from the previous example, here are some sample outputs; $\eta(\alpha, 1) = 2$, $\eta(\alpha, 2) = 1$. When modified with the function η, function *res* becomes:

$$\text{res}\left(\alpha^{(a)}\right) = \left(\sum_{k=1}^{l} f_k \sum_{i=1}^{n} \left(t_{ip_ik} \cdot s_{\eta(\alpha,i),i,k}\right)\right) \cdot \rho\left(\alpha^{(a)}\right) \tag{3.23}$$

And after updating the cost function, the last and final form is:

$$w\left(\alpha^{(a)}\right) = \left(\sum_{k=1}^{l} f_k \sum_{i=1}^{n} \left(t_{ip_ik} \cdot s_{\eta(\alpha,i),i,k}\right) + f_c \sum_{i \le j} k_{ij} \cdot c_{p_ip_j}\right) \cdot \text{ctr}\left(\alpha^{(a)}\right) \tag{3.24}$$

Since the synergy factor array is the second and final approximation of this model. Since it is very hard to measure it in practice, it is suggested that it is obtained by experimental experience or by an expert with good knowledge about the platform. However, since it is an approximation it will introduce a level of uncertainty in the model, a software architect can choose whether to use it or leave it out of the final cost function.

With this, the model $\mathbb{M}_\alpha$ is complete, and based on all the information contained

within it, the allocation decision can be made.

3.7 Summary

This chapter presented the component allocation model $\mathbb{M}_\alpha$ which contains the information about the heterogeneous computing platform, physical and architectural defined constraints, allocation function, and a set of parameters necessary to make a multi–criteria cost function for comparing performance of different component allocations. The component allocation model is defined as
$\mathbb{M}_\alpha = \left\{\mathbb{H}, \alpha, (\mathcal{T}, \mathcal{R}, \mathcal{C}, \mathcal{K}, \mathcal{B}), (\mathcal{D}, \mathcal{Y}, \mathcal{X}), M_c, F, \mathcal{S}, w\right\}$, where:

$\mathbb{H}$: is a heterogeneous computing platform, which consists of a set of software components $\mathcal{C}$, and a set of computing units $\mathcal{U}$,

α : is an allocation function which which maps a set of software components to a set of computing units, i.e. $\alpha : \mathcal{C} \to \mathcal{U}$,

$(\mathcal{T}, \mathcal{R}, \mathcal{C}, \mathcal{K}, \mathcal{B})$: is a set of arrays which contain the information about resource requirement and availability, with $\mathcal{K}$ being an approximation,

$(\mathcal{D}, \mathcal{Y}, \mathcal{X})$: is a set of matrices which contain the information about architectural constraints,

M_c : is a resource pairwise comparison matrix,

F : is a trade–off vector, provided by AHP which contains importance weights for each resource,

$\mathcal{S}$: is a synergy effect approximation trade–off array

w : is an allocation cost function

As such, the model $\mathbb{M}_\alpha$ provides all the information necessary to attribute each feasible allocation of software components onto a heterogeneous (or homogeneous for that matter) computing platform, with a number. This number enables the comparison different allocations. An allocation with the lowest non–zero cost is also the best. The idea behind this model is to use it in an early design phase to provide a software architect with a performance insight of future system. With this information, architectural decision making is largely simplified.

The procedure in which a software architect can obtain the best allocation consists of four steps:

I: Define the heterogeneous platform $\mathbb{H}$, obtain the information about resource requirements and availability $(\mathcal{T}, \mathcal{R}, \mathcal{C}, \mathcal{K}, \mathcal{B})$.

II: Define the architectural constraints $(\mathcal{D}, \mathcal{Y}, \mathcal{X})$ and synergy effect trade–off array $\mathcal{S}$

III: Perform a pairwise resource comparison and calculate the resource importance trade-off vector F

IV: Find an allocation $\alpha^{(a)}$ such that it is the lowest non–zero solution of the cost function w, i.e. $\min\left(w\left(\alpha^{(a)}\right)\right) > 0$

Henceforth, steps defined from I to IV for obtaining the optimal allocation of software components to a heterogeneous computing platform will be referred as *I-IV allocation framework*.

The following chapter presents the measurements on a real–world system which are necessary to collect all the information about the model. This information can then be used to determine the best allocation for that system and also presents the second research goal **RG-2** of this thesis.

CHAPTER 4

MEASUREMENT

This chapter presents the collection process and an analysis of extra–functional properties for a component based software architecture deployable on different computing units within a heterogeneous computing environment. The main extra–functional properties in focus of this research are the average execution time and the average power consumption. The first is traditionally identified as a system quality which reflects the overall performance, while the latter is becoming increasingly important in both everyday industrial/office and embedded computing.

4.1 Environment – the heterogeneous platform

The design of present–day embedded systems and the accompanying software is becoming ever more complex, and with an emergence of specialized computing units for accelerating certain operations, this trend is growing. Along with the benefits, this also carries some drawbacks, i.e. additional complexity which generates a variety of (side–) effects that are hard to ignore while designing a software. With this issue of growing, and sometimes even referred as unnecessary complexity, research in cyber-physical systems is gaining popularity. From the cyber-physical systems theory perspective [89, 120], software is characterized by its physical properties, which are apparent in its requirement and consumption of resources, e.g. time, energy, generated heat, etc. [121]. For software architects this presents additional considerations in the software design process. To deal with this additional complexity and considerations, this research exploits the benefits of component based software engineering (CBSE). It facilitates techniques for expressing system characteristics in the form of extra–functional properties [26], based on which software architects perform software design decisions.

The extra–functional properties which will be addressed from this point onward are the following:

a) *average execution time*, as it reflects the temporal behavior of a certain implementation of the heterogeneous platform. This research, in particular, deals with platform–dependent average execution time, as a most important system property related to the system performance.

b) *average power consumption*, as it directly affects major system design decisions, e.g. the size of the power supply, design of the cooling system, number of voltage regulators, etc. All of which influence on the physical size of the system, the processing power it can contain, durability, reliability, etc. [74].

By no means are the previous two extra–functional properties the only ones, or the most important ones which can be included in the $\mathbb{M}_\alpha$ model. However, for the reasons stated above, these were selected as particularly interesting ones for this thesis. In addition, they are selected due to their their interesting reflection of the cyber–physical theory, mainly thinking of the physical footprint exhibited by a software system. The procedure of evaluating extra–functional properties of that kind is also refereed to as *profiling*. Since this research deals with software components, it will be refereed to as *component profiling*. All the components are profiled on the same computing platform, a tracked robot which features three computing units of different kinds, a CPU, a GPU and an FPGA. Its mechanical, electrical and software configuration are described in the following section.

4.1.1 Hardware

Mechanics

The computing platform ($\mathcal{U}$) used in this research is embodied within a tracked robot named TiWo. It is built for the purpose of component profiling for this research due to the lack of commercially available robots which have multiple processing units of different types. It's mechanical design features:

- *a steel chassis*, one of the greatest challenges in designing a robot is to minimize its weight, while preserving the strength necessary to carry on board electronics. After a several failed trails, 1.5 mm steel was chosen as it had the best structural strength–to–weight ratio.
- *geared stepper motors*, chosen due to their torque, precision and ease of control by a software. They provide enough torque to overcome the rolling resistance of the robot maximally weighting 20 kg. TiWo has two Nema 23 steppers with planetary gear boxes (with 4:1 ratio), with 20 Nm of torque, weighing 1.3 kg.
- *an industrial grade plastic tracks*, since it is hard to get hold of a custom made rubber tracks, the ones which are currently in use are two sided timing belts made from industrial grade plastic. This provides a good grip on rough surfaces. On indoor smooth surfaces, some rolling may occur occasionally.
- *industrial grade plastic bearings*, metal bearings are avoided due to their weight.
- *3D printed sprockets*, the original design had aluminum sprockets, but because their weight, they were replaced by the 3D printed ones.

Weight was one of the most challenging issues of the mechanical design. Initially, the total weight of the mechanical parts (chassis, sprockets, tracks and stepper motors)

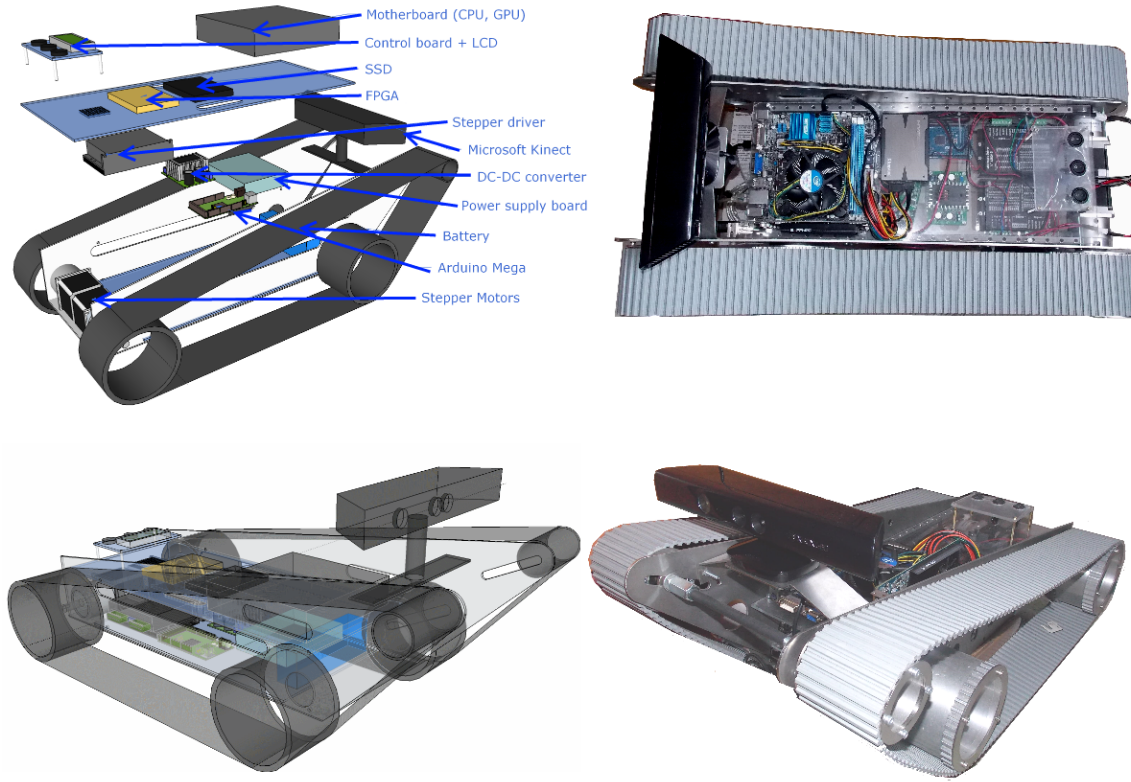

Figure 4.1: TiWo – design phases

without electronics was 17.8 kg. However with the chassis redesign, the weight was reduced by 77%. Also, 3D printed sprockets are 92.5% lighter than the original aluminum ones. Considering these two design changes, the total weight was reduced by 53%, more precisely, down to 8.3 kg.

Electronics

The computing platform is composed completely with commercial–off–the–shelf (COTS) components, and since the robots in the real–world scenarios (e.g. Mars rovers, mining robots, robot submarines, etc.) rarely have access to a power supply and use batteries, the components for TiWo were selected to minimize the power consumption. The computing hardware consists of the following:

- *CPU*, Intel i3–3240 with 3.4 GHz, with two processing cores and hyper–threading capability which enables four simultaneous threads.
- *GPU*, Sapphire Radeon HD7750 low profile graphics card with 1 GB of RAM memory.
- *FPGA*, Xilinx Spartan 6 (LogiPi board[1]).
- *motherboard*, mini-ITX Asus P8H61-I.
- *memory*, Kingston HyperX Blue 8 GB DDR RAM.

[1]LogiPi board, a Kickstarter FPGA project for RaspberryPi – http://valentfx.com/logi-pi/ accessed in period between fall 2013, spring 2016

- *storage unit*, Kingston HyperX Blue solid state drive 240 GB.
- *vision system*, Microsoft Kinect Sensor (1^{st} generation).
- *motor control*, SainSmart M542 Microstepper motor driver, Arduino Mega for high level motor control and communication with low–level hardware.
- *power supply*, M4 ATX intelligent power supply plugged into a laboratory 6–30 V power source or 5 cell LiPo battery (Turnigy 5000 mAh, 6S battery).
- *misc*, 2x DC/DC converters, 2x16 LCD display.

In this list, three components are of particular interest for this research; the CPU, the GPU and the FPGA. These three electronic components are the main on–board computing power, i..e the heterogeneous computing platform. These are used in the following sections for the analysis of software allocation alternatives. Considering the component allocation model $\mathbb{M}_\alpha$, the CPU, the GPU and the FPGA are the elements of the set of computing units, $\mathcal{U}$.

4.1.2 Software

The software components used for the analysis of allocation choices and verifying the allocation I-IV framework are developed to work properly on both Windows and Linux. However, for the purpose of component profiling and experimenting in this research Linux Ubuntu 14.10 LTS was used. The following six components are used for profiling:

- *Image input component (II)*
  - *Function:* It is in charge for taking images from any available imaging device present in the system. Currently, the imaging device is Microsoft Kinect, and therefore the component provides the functionality to load a) plain color (RGB) image, b) infrared (IR) image, c) depth image and finally d) the combination of IR and RGB image. The key implementation assumption of the Image input component is the support of infrared and stereo vision.
  - *Allocation:* The architectural constraint, due to the implementation of this component, is that it can only be allocated on a CPU.
  - *Implementation:* This component is implemented in Java, using a OpenKinect library which makes it work properly on both Windows and Linux.
- *Image filtering component (IF)*
  - *Function:* It is probably the most important component due to its processing ability. It accepts messages with two parameters, the first is the filtering chain setup and the second one is the image to be processed. Currently it handles five filters; a) a gaussian blur filter, b) a sobel filter, c) an erode filter, d) a dilate filter and e) a hysteresis filter. Each filter can be used individually or several of them can be combined in a filtering chain (implemented as a pipeline).

- *Allocation:* this component can be allocated on all three computing units, i.e. on the CPU, the GPU and the FPGA (with some limitations further described in the following section).
- *Implementation:* This component is implemented for several different processing paradigms, all of which use a multi–layered approach. For the CPU and GPU implementation the bottom layer was implemented in C++. OpenCL was also used to implement filtering kernels which could be executed on both the CPU and the GPU (without any changes). The intermediate layer is a Java native interface (JNI) wrapper which handles the communication between the top and the bottom layer. The top layer is implemented in Java, and provides functions to simply create the filtering chain out of the available filters and to determine the execution environment (CPU or GPU). The implementation for the FPGA is different, it consists of four layers. The bottom layer is implemented in VHDL. The second layer is in charge for communication with the FPGA device and it is written in C++. The third layer is the JNI wrapper which communicates with C++ code of the second layer, and with the Java code of the fourth layer. The third layer implements the communication between the RaspberryPi (which contains the FPGA add–on board) and the CPU. The communication is handled via the Ethernet using the UDP protocol. The final layer is a high level layer which abstracts all the hardware below. A software developer simply defines the filters which are supposed to be used and an image on which they should be applied. After calling the image filtering functions, all of the previously described layers are used to deliver the processed image.

– *Mission manager component (MM)*

- *Function:* The component is in charge for implementing a simple parser which handles the robot mission files. Mission files describe what a robot should do using a simple syntax, e.g. (`START`, `FORWARD`, `DETECT`, `UNTIL`, `OBJECT_DETECTED`, etc.). This component is used in combination with the Main control component and it waits for the signal to load the mission file. The Main control component then uses the interpretation provided by this component and communicates with the Actuator control component in order to execute the mission.
- *Allocation:* This component can be allocated only on the CPU.
- *Implementation:* Mission manager component is completely implemented in Java.

– *Main control component (MC)*

- *Function:* This component binds all other components together to form a functioning whole. It initializes the communication channels between the components and queues the Mission manager component to start with the

mission execution. This component also provides a GUI.
    - *Allocation:* This component can be allocated only on the CPU.
    - *Implementation:* It is implemented entirely in Java and it uses several different communication libraries like JSSC and Kryo along with the Java Swing components for the user interface.
- *Object detection component (OD)*
    - *Function:* It is one of the most important components. As the name suggests, this component is used for detecting objects in the provided image. For the purpose of this research, a custom Haar classifier was created to detect arrows which show the robot the direction in which it should proceed. The same classifier (without code rewriting) was tested on multiple computing units.
    - *Allocation:* This component can be allocated to the CPU and the GPU.
    - *Implementation:* This component is separated in to three layers. The bottom layer is written in C++, and uses OpenCV accelerated with OpenCL. This layer uses the Haar classifier to detect the object in a provided image. The result is passed to the upper layer through the intermediate layer, i.e. a JNI wrapper. The higher level enables a software developer to select the desired classifier and an image.
- *Actuator control component (AC)*
    - *Function:* The role of this component is to scan the ports searching for the interface with the lower level software. When found, it creates a messaging environment and exchanges commands from the Main control component to the lower level software. The lower level software is Arduino C code which controls the stepper motors. This component also provides abstractions of the low level commands (e.g. instead of dealing with binary coding for the steps of the stepper motor, a developer simply sends the `FORWARD` message and the lower software level deals with the controller signals).
    - *Allocation:* This component can be allocated only on the CPU.
    - *Implementation:* The component is implemented in Java. It is multi–threaded to handle the two–way full duplex communication.

Figure 4.2 shows the simplified software architecture layout, and as one can notice, not all the components communicate. The Main control component binds the functionality of all components into a functioning whole.

Only two components can be allocated on computing units different than CPU. The Image filtering component can be allocated on the CPU, the GPU and the FPGA, while the Object detection component can be allocated to the CPU and the GPU. Although this might be a minor drawback, the important point about the implementation of Image filtering component is the fact that each filter contained within this components can be treated as a component on its own. Filters are merely grouped together in the same

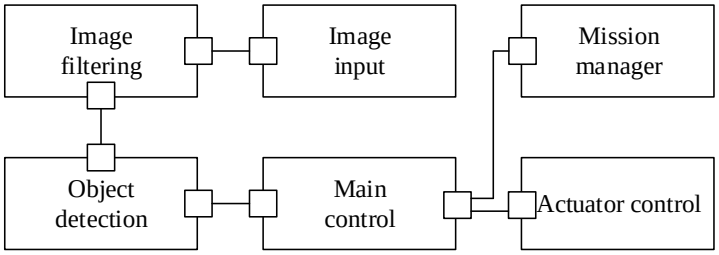

Figure 4.2: Simplified software architecture layout

component since they share the similar functionality and the same interface. Having that said, Image filtering component consists of five sub–components, which can be separated and individually allocated. For the time and power profiling in this chapter however, it will be treated as a single component. Therefore, the set of software components $\mathcal{C}$ for the model $\mathbb{M}_\alpha$ has six, previously listed components.

4.2 The measuring procedure

For every software component described in the previous section two main properties are measured; the *average execution time* and the *average power consumption* down to the level of individual operations performed by the components. This section describes and clarifies the measurement procedure for the component profiling.

4.2.1 The average execution time

The average execution time of the components was measured and recorded by the custom software which was used for all the components. The elapsed time, i.e. execution time of a certain operation available on each component was calculated by recoding the starting and the finishing time of the operation. Although there were some issues related to faster operations (measured in nanoseconds), these issues were handled and described in this section. For each of the computing units the procedure was similar but a bit different, for the;

a) *CPU*, Figure 4.3 shows the points at which the CPU average execution time was measured. All the components are accessible from Java code, and therefore, for every component the execution time was measured in Java code (`System.nanoTime()`). The figure shows A as the starting point of time measurement, while B is the time when the operation is finished. The time $B - A$ represents the elapsed time, i.e. the execution time of a certain operation (service) provided by a component.

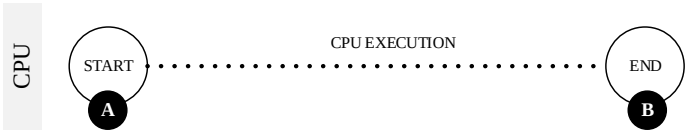

Figure 4.3: Time measurement points for the CPU

b) *GPU*, a GPU component cannot be executed without using a CPU, so the measurement points should include both of these computing units. Consider the ones shown in Figure 4.4. Respectively the points A and D are the starting and ending point of executing a operation on the GPU. The points B and C are the starting and ending time of executing a GPU kernel. $C - B$ is the execution time of a GPU kernel, while $D - A$ is the total execution time necessary to send and obtain data processed by the software component allocated to the GPU. Therefore, $(B - A) + (D - C)$ is the data loading time, i.e. the time necessary to transfer the data to and from GPU. Time was measured in both Java and C++ code.

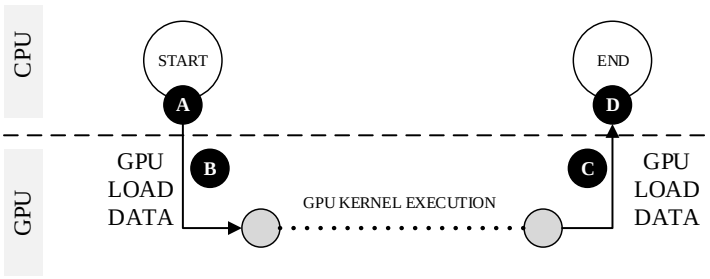

Figure 4.4: Time measurement points for the GPU

c) *FPGA*, the average execution time was measured similarly as for the GPU. Figure 4.5 shows the measurement points. As previously, both the execution time and the data loading time were measured. Regardless to one extra computing unit between the CPU and FPGA, there were four measurement points. $D - A$ is the total execution time (including the data transfer) of the operation on the FPGA, while $C - B$ is only the execution time necessary for the FPGA to finish an operation. Hence, $(B - A) + (D - C)$ is the data loading time. All the time measurements were performed in both Java and C++ code.

The measured execution time(s) of the operations provided by all the components were stored in a structured file for later processing. For the three components (all of them written fully in Java; Mission manager, Main control and Actuator control), using Java commands to record the operation starting and finishing time was not adequate. Some operations were executing to fast to draw any meaningful conclusions from the data, since the elapsed time, fairly often turned out to be negative. According to Java documentation, the previously mentioned `System.nanoTime()` method *"provides nanosecond precision, but not necessarily nanosecond accuracy"*, and also *"no guarantees are made*

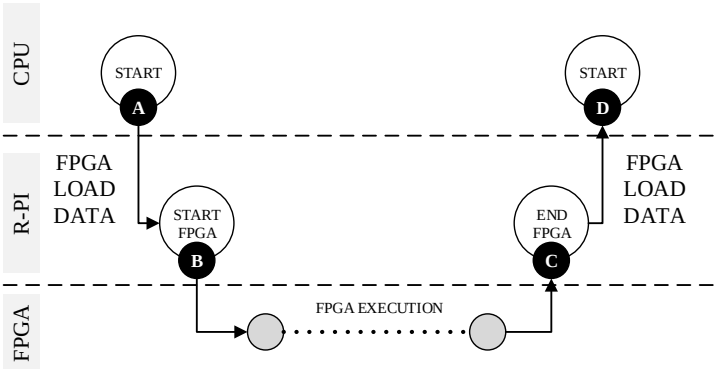

Figure 4.5: Time measurement points for the FPGA

about how frequently values change"[2]. Therefore, a special tool was used to profile operations with extremely fast execution time, VisualVM. It is a tool to monitor and troubleshoot Java applications. Among many features, it can attach to any running Java application and perform time profiling. The level of precision is down to a single operation, with minimal overhead on monitored application[3].

4.2.2 Average power consumption measurement

In order to perform the power consumption profiling of the software components, a high–precision multimeter was used; GwInstek GDM–8342. Other than the precision its most important features include recording measurements external USB storage units, controlling the sampling rate and simultaneously measuring the current and the voltage. Once the device all is setup for measurement and the statistical number of necessary samples is calculated, it is very important to determine the measurement points. Figure 4.6 shows several different measurement points for the computing units:

a) *CPU*, is connected to the motherboard via the socket with several hindered pins which makes it inaccessible for direct current measurement. Therefore, the current was measured at the point A (Figure 4.6), right at the power source. While this point provides the power consumption for the entire motherboard, a simple technique by Collange et.al. can be used to obtain only the CPU's power consumption ([23]). This is further discussed in the coming sections.

b) *GPU*, initial measurements of power consumption for the GPU was performed using a PCI–Express pull–up board, i.e. the point B. However measurements have shown that this point exposes only the 3.3 V rail without the 12 V rail. Since the majority of the power goes through the 12 V rail, this point wasn't acceptable. Another option involved using current clamps however there is a questionable measurement precision of this method. Finally, the measurements took place at

[2]JavaDoc, http://docs.oracle.com/javase/6/docs/api/java/lang/System.html#nanoTime accessed in June 2015

[3]VisualVM http://visualvm.java.net/, accessed June 2015

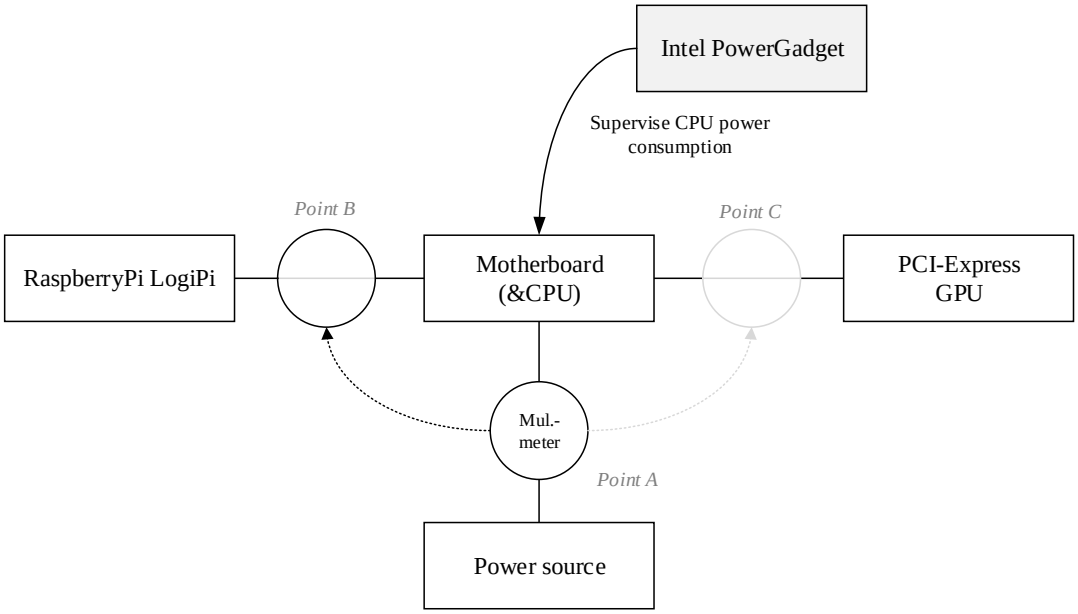

Figure 4.6: Measurement setup

the point A, as for the CPU.

c) *FPGA*, since it was on board a Raspberry PI, it can be supplied via a USB cable. Therefore, some modifications to USB cable were made which provided the access to the series for current measurement. As the Figure 4.6 shows, the power consumption was measured at the point B.

Besides the high–precision multimeter, Intel's Power Gadget tool was used to additionally monitor the power consumption. It is a software tool provided by Intel which uses low level software hooks to approximate the power drain. However, the measurements have shown that this tool is very imprecise and cannot be used to augment the data recorded by the multimeter. Instead, it was used to monitor the CPU activity while the idling parameters were measured for other computing units. For example, while measuring GPU power consumption, using the Power Gadget the activity of the CPU was monitored to make sure it is idling so it doesn't affect the measurement. In addition to Intel's Power Gadget, `htop` was also used for the same purpose.

Considering the measurement points, an obvious question is how to measure only the power consumption of the CPU without any other disturbance. Do do that a similar approach was used as suggested by Collange et.al. in their paper on the power consumption from a software perspective. The main idea behind this goal is to measure the CPU's average power consumption with as little load as possible, i.e. measure the idling average power consumption. Having the CPU idling average power consumption, if one wants to obtain the average power consumption of a certain software, it is necessary to execute this software and measure and detect for how much does the average power consumption increase above the idling power consumption. Finally, the power consumption of software being profiled is the difference between the average power

consumption measured while it was executed and the idling average power consumption of a computing unit. However, since CPU is prone to running other tasks, a certain modifications of OS could be made to make it more predictable, and the measurements must be repeated multiple times.

Although it seems simple, it is far from it. During the measurements for this research, the CPU was continuously supervised not to execute other tasks, and all the unnecessary operating system tasks were killed and reduced to only the essential ones. However, once getting the large enough sample of the idling power consumption and comparing it to the sample collected while iteratively executing the software components in a controlled manor the results were substantial. The exact same method was used for the GPU.

Considering this method, one can quickly come to the conclusion that the power consumption data acquired by this way is not 100% accurate, meaning that it contains some amount of overhead generated from the supporting hardware infrastructure, namely the motherboard, the memory and the storage. However, the power consumption of the solid state drive is very low, so for this research it could be left out. Also, a CPU without RAM, and a motherboard can hardly operate by itself, therefore it could be argued that it is welcoming to consider the entire supporting hardware infrastructure of the CPU in such measurements.

As for the GPU related measurement, knowing the idling power consumption of the CPU it is easy to extract the power consumption of the GPU alone (while measuring at the point A Figure 4.6). Similarly as for the CPU, for the GPU it is also welcoming to measure the power consumption of the entire supporting hardware infrastructure (at least, for this research it is), which is the reason behind using the measurement point A. And as the idling power consumption of the CPU was subtracted from the GPU power consumption measurement, it could be noticed that the GPU cannot really work without the CPU. Therefore, when a component is allocated to the GPU, it will always leave a power consumption footprint on the CPU also. Such components in reality use two computing units and influence on their power consumption, so for their true power consumption it is advisable to monitor both computing units. It was done this way in this research. While the CPU's idling power consumption was not attributed to software components allocated on the GPU, the extra load of the CPU in such cases was considered. Figure 4.7 shows how can one recognize the physical footprint a software component leaves on the power measurement curve of a computing unit. It is easily to distinguish between the idling computing unit and the computing unit under a load. When measuring the real average power consumption the data inside the yellow box was used, while the rest represents the intentional idling of the unit and it was dismissed. This idling was intentionally built in so that the raise of the power consumption could be detected more efficiently.

The particular case of Figure 4.7 shows the power consumption of a GPU executing the

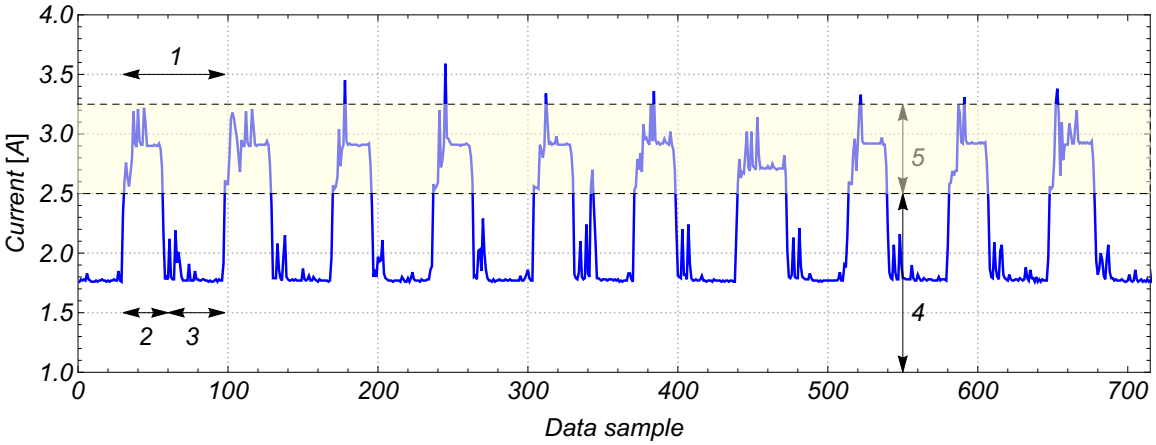

Figure 4.7: Measurement sample, 1 – function call, 2 – GPU active, 3 – intentional idle, 4 – discarded data, 5 – area with real average power consumption (yellow box).

Image filtering component in a controlled manor; ten operations executed in a row with 500ms of delay in between. Notice also that every power raise (e.g. the beginning at point 1) starts with a small peak. This is attributed to the activity of both CPU and GPU (2), i.e. loading the data to the GPU. Also, a similar occurrence can be noticed after the operation is complete (3).

4.2.3 Measurement steps and parameters

The measurement steps necessary to acquire the average execution time and the average power consumption of the components presented in subsection 4.1.2 are the following:

1) *Initial measurement*, before recording the real data which is used to draw conclusions, an initial sample was taken. This sample was processed and analyzed to find out how does the measured characteristic of the component behave,
2) *Initial verification of the standard deviation and the confidence interval*, the initial data sample was analyzed using Minitab[4] to determine the standard deviation and the confidence interval that could be expected for the real data mean,
3) *Calculating the necessary number of samples*, based on the initial measurement the number of samples necessary to satisfy the selected confidence interval of 95% was determined. The confidence interval of 95% was selected under the assumption that it will reflect a good enough estimation of the measured values,
4) *Performing the real measurement*, once the number of samples was known, the repeated measurements took place(more details on this are given in the following subsections),
5) *Statistical analysis of the measured data*, using Minitab the collected data was analyzed to acquire the mean value, standard deviation and to perform the t–test. In

[4]a statistical tool, available at https://www.minitab.com/, accessed through the summer and winter of 2015

one case, data normality was also verified (one data set with less than 30 samples) The next subsection presents all the parameters of the measurement, along with the guiding remarks for anyone with a desire to repeat the measurements and recreate the results.

Measuring idle current and voltage

Before proceeding to power profiling of software components, as it was described previously, it is first necessary to obtain the average idling power consumption for each of the computing units. To do this, for each computing unit, all the unnecessary processing tasks were killed and reduced to bare essentials. For the GPU this meant reducing the graphics operations of the operating system to minimum (e.g. reducing mouse movements and performing GUI operations, basically reducing all operations which might cause the Linux X–system to redraw the screen). For the CPU this meant killing the unnecessary processes, e.g. auto update, background servers, daemons, etc. As for the FPGA, this meant only to not load the program to the FPGA memory, since by default it doesn't handle any operations until it is explicitly initiated.
The results of measuring idling power consumption for the computing units at hand are the following:

a) *CPU*, initial measurements have shown that the current ant the voltage of the idling CPU have stable values with low standard deviation. There were 4900 samples taken for both the current and the voltage at 10 Hz (10 samples per second), while the GPU and the FPGA were not connected to the system.
b) *GPU*, similarly as for the CPU, the GPUs voltage and current consumption was stable with low standard deviation. The measurement performed at the points A and C (Figure 4.6) with 4900 samples at the 10 Hz rate. 30–40% of the current was flowing through the 12 V rail which could not be accessed through the point C. Therefore, the point A was used, with FPGA absent from the system.
c) *FPGA*, used an enhanced 5 V, 2 A USB power supply. Since the LogiPi board (FPGA) is connected to the robot through the RaspberryPi, two measurements were made to get the maximal precision of the average idling power consumption. Both measurements were performed with 1550 samples at 10 Hz sampling rate. The first measurement was without the LogiPi board present in the system while the second one was with the board present in the system and with binary file loaded (VHDL executable), but with no processing data. The average idling power consumption was the difference between these two measurements.

Throughout the measurement, a stable power supply was used (Codegen 400W, model 300XX) so there were no major voltage drops and in the remaining measurements there was no need to simultaneously measure both power and voltage. In later measurements, only current was measured with higher sampling rate.

Having obtained the average power consumption of computing units in ther idle state, the next step is to measure the changes which occur while processing data.

Measuring component performance

For every component, the average power consumption and the average execution time were thoroughly, thoughtfully and carefully measured and recorded. This section presents the measurement parameters which were used for component profiling. The parameters are organized into structured tables containing the following information:

a) *Number of runs*, the number of repeated calls of the same operation within the component.
b) *Sample size*, contains the information about the number of samples which were recorded for a particular component. It varies depending on the duration of the performed operation and previously determined standard deviation, and therefore the value in the table can contain both the minimal and maximal number of samples which were taken for the particular component.
c) *Usable sample size*, the recorded samples contain extra data which should be removed, e.g. known outliers, extremes, intentional idling between operation calls etc as shown in Figure 4.7. So this row shows the number of samples which are usable for future analysis.
d) *Sampling rate*, the number of samples per a time unit, i.e. sampling frequency (it is only applicable for the measurement of the current).
e) *Repeated measurements with different parameters*, some components are capable of handling different parameters which means that their load is different, e.g. different image size, mission file content, etc. This row shows number of repeated measurements for the same component but with a different data inputs.
f) *Remarks*, miscellaneous comments related to the measurement.

The measurement parameters for all the components

Table 4.1 shows measurement parameters for the Image filtering component and the case when it is allocated to the CPU. The sample size, depending on the input image was between 3000 and 8000. However all these samples were not usable, so after initial processing, the number of usable samples was between 500 and 4420. Table 4.1 presents the data for both the average power consumption and the average execution time.

Measuring the current resulted in 30 files, on average containing 4000 data samples, which were then processed and filtered. The measurement of execution time also resulted in 30 files, each containing 50 rows of data. After the initial processing (with custom Java programs) of both current and execution time, the data was analyzed in Minitab.

Table 4.1: Measurement parameters for the Image filtering component, allocated on the CPU

Image filtering component - CPU	Current	Time
Number of runs	50	50
Sample size	3000 – 8000	50
Usable sample size	500 – 4420	50
Sampling rate	40 Hz	–
Repeated measurements with diff. param.	30	30
Remarks		
Before each measurement, there was a 5 second pause time, which was necessary for the current to stabilize (after setting up the measurement environment), and between each run (call) there was 0.5s of intentional delay. The current measurement setup was for up to 10A, i.e. 3 decimal points of precision. Each filter setup was measured 5 times with different input image size: a) QVGA 4:3 320×240 , b) VGA 4:3 640×480, c) SXGA 4:3 1280×960, d) FHD 16:9 1920×1080, e) 8KFD 1:1 8192×8192. The resolutions QVGA–HD were selected due to their popularity in the image processing community, except for the 8KFD resolution which was selected to test the extreme inputs, i.e. to yield the maximum processing power of a computing unit. Since the components could be arranged to perform different filtering chains, the following configurations were examined: (Sobel), (Gauss), (Sobel, Gauss), (Sobel, Gauss, Erode), (Sobel, Gauss, Erode, Dilate), (Sobel, Gauss, Erode, Dilate, Hysteresis).		

Table 4.2 shows the measurement parameters for the same component, but allocated on the GPU. The number of samples was smaller than previously because the operations were on average performed faster. The measurement resulted with 30 files, on average containing 2000 rows of the current related data, and with 30 additional files containing 50 rows of time related data. These 60 files were also parsed by a Java program and analyzed in Minitab.

Table 4.2: Measurement parameters for the Image filtering component, allocated on the GPU

Image filtering component - GPU	Current	Time
Number of runs	50	50
Sample size	1500 – 3900	50
Usable sample size	400 – 1800	50
Sampling rate	40 Hz	–
Repeated measurements with diff. param.	30	30
Remarks		
The same setup as for the CPU, see table Table 4.1. This procedure included measuring both the data loading time and the data processing time.		

Table 4.3 shows the measurement parameters for the Image filtering component allocated on the FPGA. As for the CPU and GPU, the measurement resulted in 60 files which were then parsed in Java and analyzed in Minitab. Number of samples for measuring the current was not deviating as much as for the CPU and the GPU (which is further dis-

cussed in the following section), and usable sample for measuring time was not always the same to the number of runs. In some cases, the operation duration was reported as a negative value, which was due to C++s' inability to guarantee time measurement accuracy. These values were dismissed. But since the time was measured on both C++ and Java layers, Java recorded the total execution time (data transfer and processing) which was always usable. In addition, the measurements for FPGA were repeated only six times. Once for each filter setup only for the QVGA resolution, since the IP core only supported QVGA.

Table 4.3: Measurement parameters for the Image filtering component, allocated on the FPGA

Image filtering component - FPGA	**Current**	**Time**
Number of runs	550	550
Sample size	4100 – 4300	550
Usable sample size	470 – 610	390 – 550
Sampling rate	40 Hz	–
Repeated measurements with diff. param.	6	6
Remarks		
Although the general remarks are the same as for the previous two measurements, an image input of only one resolution was used, i.e. QVGA. In order to process multiple resolutions, the FPGA would need an IP core which provides the option to change the resolution, but unfortunately, LogiPi provides only an IP core for the QVGA resolution. Similarly as for the GPU, for the FPGA there were two measuring points for the execution time. The first one at the LogiPI which is equivalent to the processing time, and the second one in the main Java program, which also includes the data transfer time, as explained previously.		

Table 4.4 shows the measurement parameters for the Object detection component, allocated on the CPU. Similarly as for the Image detection component, there was a five second waiting time before the start of each measurement, which was enough for the current to stabilize. Also, before each run there was a one second delay to make data analysis easier later on. This measurement resulted with 24 files, 12 for both the current and the time data, which was parsed and analyzed further.

Table 4.4: Measurement parameters for the Object detection component, allocated on the FPGA

Object detection component - FPGA	**Current**	**Time**
Number of runs	250	250
Sample size	5000 – 7000	250
Usable sample size	2300 – 3500	250
Sampling rate	40 Hz	–
Repeated measurements with diff. param.	6	6
Remarks		
To perform the object detection, a custom Haar classifier was used. It was trained using OpenCV with OpenCL extensions. The resulting classifier is an XML file which is used by C++ code to classify, i.e. to detect an object. The same XML file was used for both CPU and GPU detection. The object in question is an arrow (points the direction in which robot should go). For each measurement, three different images were used, two containing, and one without an object to detect in two different sizes, 640×480 and 1920×1080.		

Table 4.5 shows the measurement parameters for the Object detection component allocated on the GPU. As previously, the measurement resulted with 24 files which were parsed and further analyzed in Minitab.

Table 4.5: Measurement parameters for the Object detection component, allocated on the GPU

Object detection component - GPU	**Current**	**Time**
Number of runs	250	250
Sample size	5500 – 5800	250
Usable sample size	1300 – 1450	250
Sampling rate	40 Hz	–
Repeated measurements with diff. param.	6	6
Remarks		
The measurement setup and remarks are the same as for the CPU, see table Table 4.4		

The measurement parameters for the Image input component, allocated on the CPU is shown are Table 4.6. Measurements resulted with 8 files, 4 of which were related to current data, with average of 10000 rows, and other 4 related to time data with 500 rows. In each run, image of the same resolution was acquired, however with different method; RGB, infrared, depth or combined.

Table 4.6: Measurement parameters for the Image input component, allocated on the CPU

Image input component - CPU	**Current**	**Time**
Number of runs	500	500
Sample size	9000 – 13000	500
Usable sample size	4800 – 8900	500
Sampling rate	40 Hz	–
Repeated measurements with diff. param.	4	4
Remarks		
For this component four different repeated measurements were performed; a) normal RGB image, b) infrared image, c) RGB depth image, d) infrared and RGB image combined. The maximal supported resolution for the hardware at hand (Microsoft Kinect) was 640×480 px.		

The measurement for the Main control component, which can be allocated only on the CPU is shown in Table 4.7. This is also the first component for which the time was measured using the VisualVM. Measuring the average power consumption for this component was very exhaustive, since the operation needs to last long enough to be detected by the multimeter, and the operations within this component were very short. To detect current changes a large sample size was needed with 10000 repeated runs.

Table 4.7: Measurement parameters for the Main control component, allocated on the CPU

Main control component - CPU	**Current**	**Time**
Number of runs	1000 – 10000	1000 – 10000
Sample size	200 – 2500	1000 – 10000
Usable sample size	50 – 2000	1000 – 10000
Sampling rate	40 Hz	–
Repeated measurements with diff. param.	6	6
Remarks		
For this component there were six repeated measurements, i.e. one measurement for each of the most common operations. Time measurements were performed using VisualVM.		

Table 4.8 shows the measurement parameters for the Actuator control component allocated on the CPU. Similarly as for the previous component, the average execution time for this one was also measured using VisualVM. In order to detect any changes in the current measurement, the sample size varied between 2000 and 10000.

Table 4.8: Measurement parameters for the Actuator control component, allocated on the CPU

Actuator control component - CPU	**Current**	**Time**
Number of runs	2000 – 10000	2000 – 10000
Sample size	160 – 770	2000 – 10000
Usable sample size	29 – 301	2000 – 10000
Sampling rate	40 Hz	–
Repeated measurements with diff. param.	3	3
Remarks		
For this component three repeated scenarios were measured. Since this component deals with parsing and transferring messages to the lower–level software, each possible message was used. The component was set up in a way to change messages for each run.		

The measurement for the final component is shown in Table 4.9. The operations performed by this component had a short duration so there was 25000 samples necessary to detect changes in the usage of current.

Table 4.9: Measurement parameters for the Mission manager component, allocated on the CPU

Mission manager component - CPU	**Current**	**Time**
Number of runs	25000	25000
Sample size	1040	25000
Usable sample size	398	25000
Sampling rate	40 Hz	–
Repeated measurements with diff. param.	1	1
Remarks		
For this component only one measurement was enough, since at an each run, there is only one basic operation used (top level operation – facade pattern). The operation deals with reading the mission file and translating it in a set of available commands. 10 different mission files were used in all of the runs (i.e. each mission file was used 2500 times).		

Considering the measurement setup presented in this section, it is evident that the measurements were performed systematically and exhaustively. The next section presents the analysis of the collected data along with its interpretation necessary to use it as an input to the $\mathbb{M}_\alpha$ model.

4.3 The results

This section presents the results of measurements along with the corresponding statistical analysis. The measurements resulted with 183 files, containing 450470 rows of raw data. These files were processed by a custom Java program and analyzed further in Minitab and Microsoft Excel. Table 4.10 shows particular details and numbers related to the generated files. The data about the Main controller, Actuator controller and the Mission manager component is absent from the table since these components were profiled with VisualVM, which does not provide raw text files. But, the raw data stored in the VisualVM file format was stored for the analytical purposes.

All the collected data within this research, along with software code is publicly available, as Open Science Data at URL-s provided on the last page of this thesis (CV page).

Table 4.10: Quick facts about the collected data

	Files		Records (raw)	
Sample size	Current	Time	Current	Time
Image filtering, CPU	30	30	94820	10920
Image filtering, GPU	30	30	82251	10680
Image filtering, FPGA	6	12	25175	82716
Object detection, CPU	6	6	37988	1572
Object detection, GPU	6	6	34184	1566
Image input, CPU	4	4	44771	2021
Main control, CPU	5	–	6620	–
Actuator control, CPU	3	–	1246	–
Mission manager, CPU	1	–	1040	–
Idle, CPU	1	–	4900	–
Idle, GPU	1	–	4900	–
Idle, FPGA	2	–	3100	–
Total	95	88	340995	109475

Each measurement result presented in this section will contain the same structure with the following statistical data; the number of samples (N), the mean value of the measured variable (Mean), standard deviation (Std.dev.), standard error (SE Mean) and the upper and the lower bound of the confidence interval of 95% (95% CI).

The meaning of the confidence index

The clarification for the confidence index is shown in Figure 4.8. $\bar{x}$ represents a sample mean, while μ represents the mean of the population, i.e. the real mean value of the measured variable. Standard error (SE), is the measure of variability or more precisely

an estimate of the standard deviation of the sampling distribution. It is calculated as $SE = \sigma/\sqrt{N}$, meaning that the larger the sample, the smaller the SE, and also the wider the -SE, SE area shown in Figure 4.8. So to get the 95% confidence interval, standard error is multiplied by 1.96, which is a widely known statistical multiplier given by the normal distribution, also known as z–table[5]. Consequently to this, it is possible to claim that the real mean value is between the upper and lower bound of the confidence interval at the level of significance of 95%. Working this procedure backwards, one can obtain the number of necessary samples in order to achieve this significance, with an initial guess of the standard deviation.

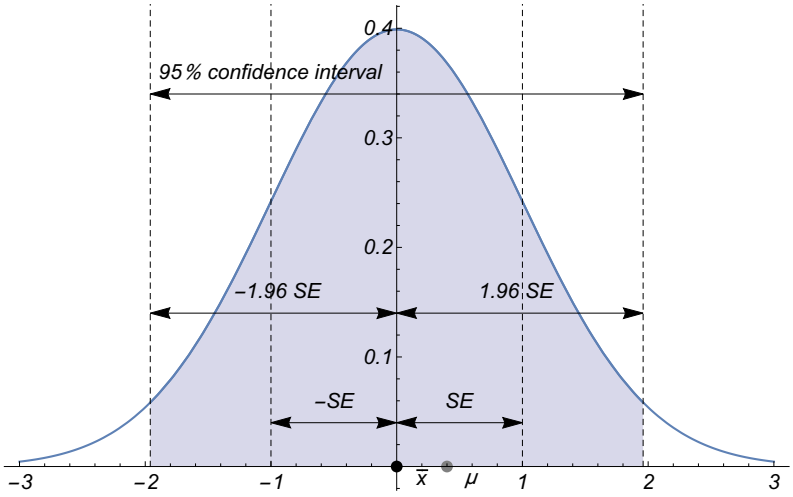

Figure 4.8: 95% confidence interval explained

4.3.1 Idle system results

Since there is no such thing as idle execution time, this section applies only to the average power consumption. It shows the results of measuring the average idle power consumption for each computing unit, i.e. the CPU, the GPU and the FPGA. These values represent the default power consumption of a running system with no particular processing tasks. Any change in these values indicates some processing, i.e. executing a software which leaves a physical footprint. The results are shown in Table 4.11.

[5]Since the population mean is unknown, t–table must be used, however for $N > 30$, z– and t– tables can be considered the same.

Table 4.11: Average power consumption of an idling system.

Variable	N	Mean	Std.dev.	SE Mean	95% CI	
CPU only						
I (A)	4151	0.783378	0.007274	0.000113	0.783157	0.783599
U (V)	4151	11.7942	0.0082	0.0001	11.7940	11.7945
P (W)	4151	**9.23935**	0.08554	0.00133	9.23675	9.24195
CPU and GPU						
I (A)	4695	1.02991	0.00619	0.00009	1.02974	1.03009
U (V)	4695	11.7704	0.0068	0.0001	11.7702	11.7706
P (W)	4695	**12.1225**	0.0730	0.0011	12.1204	12.1246
FPGA						
I (A)	1550	0.478842	0.006028	0.000153	0.478541	0.479142
U (V)	1550	4.84153	0.00462	0.00012	4.84130	4.84176
P (W)	1550	**2.31831**	0.02943	0.00075	2.31684	2.31978

Table 4.11, clearly shows that when using only the CPU, the entire system consumes 9.23 W. Not just the CPU, but its entire supporting electronic infrastructure (motherboard, memory, caches, etc.). When the CPU is not processing any data the power consumption is very stable so any data processing would significantly raise the power consumption and it could be easily detected (Figure 4.7). Furthermore, the table shows that the average power consumption of the idling GPU is 12.12 W, and of the idling FPGA it is 2.31 W. Considering that the total power consumption of the system containing only a CPU is 9.23 W, notice that when the GPU is added, the power average power consumption increases for only 2.89 W. This might seem low, but keep in mind that this is a low profile GPU. Also, the measurements have shown that when the GPU is present, the CPU is used a bit less (about 15% less as indicated by the Intel Power Gadget) since the graphics rendering is offloaded to the GPU. So, in reality the GPU does consume a bit more than 2.89 W but it is somewhat irrelevant. Rather than being interested for a precise power consumption of a particular unit, the main concern here is the average raise of the power consumption when a computing unit is under load by a software component.

All the measured values presented in Table 4.11 were statistically verified with Minitab. A typical output from Minitab is shown in Figure 4.9. It includes a histogram plot, a report on the statistical power[6] of the sample and the confidence interval comments. Since there are 185 different measurements, showing all the outputs as images would not be very useful or presentable, so instead of this all the results are organized

[6]The average power consumption values presented in Table 4.11 have the statistical power of 100% for accuracy of at least 2% around the mean.

in the tables showing: the mean value, the standard error and the confidence interval.

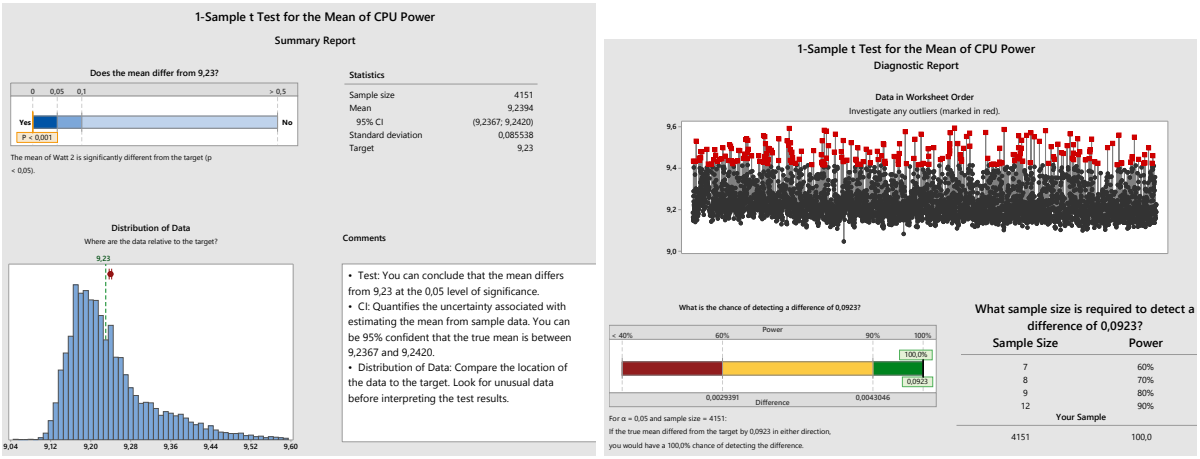

Figure 4.9: Example of Minitab t–test report

4.3.2 Software component performance results – average power consumption

Image filtering component

Table 4.12 shows by how much, on average, has the average power consumption increased above the value of idling power consumption while executing the Image filtering components with different inputs.

Here is an example of how to properly interpret the table. Consider the *S,G* row and the *QVGA* column. The value 3.0382 W means that the average power consumption of the CPU increases for 3.0382 W above the idling power consumption, while it is executing operations from the Image filtering components, with the filtering configuration *Sobel, Gauss*. The rows respectively represent the following filtering configurations: (Sobel), (Gauss), (Sobel, Gauss), (Sobel, Gauss, Erode), (Sobel, Gauss, Erode, Dilate), (Sobel, Gauss, Erode, Dilate, Hysteresis), while the columns represent different computing units; the CPU, the GPU and the FPGA, along with input images of different sizes (see Table 4.1).

Table 4.12: Image filtering component, average power consumption in Watts, for different inputs and operations

	CPU					GPU					FPGA
Resolution	QVGA	VGA	SXGA	FHD	8KFD	QVGA	VGA	SXGA	FHD	8KFD	QVGA
S	2.9639	3.3125	3.9570	5.4504	16.2547	4.6727	5.4307	5.7839	6.4830	19.9107	2.4345
G	2.9009	3.0186	3.5086	3.8794	16.7041	4.7375	5.0400	5.4460	5.6532	19.7565	2.4395
S,G	3.0382	3.2566	3.9846	7.2490	16.2969	5.8686	5.6862	6.7314	9.4641	20.5100	2.4348
S,G,E	3.1493	3.5405	4.4101	8.2373	16.4579	6.0381	5.9074	9.6826	9.9184	20.6782	2.4352
S,G,E,D	3.3123	3.6256	7.5179	9.2339	16.4832	6.1793	6.2664	9.9228	10.6593	20.6151	2.4318
S,G,E,D,H	3.3966	3.8535	7.6358	9.1231	16.7893	6.2688	6.4524	9.9566	11.4692	21.3956	2.4383

To put this data (Table 4.12) in perspective, its visualization is shown in Figure 4.10. The data is grouped by the image size for each computing unit and it reveals that:

a) *increasing the filter size does not proportionally increase the power consumption*; consider the combination QVGA, VGA for CPU and QVGA, FHD for GPU. Notice that filter SGEDH does not increase the power consumption to be the equivalent to the sum of individual filters. In the $\mathbb{M}_\alpha$ this is called a *positive synergy effect*.

b) *increasing the image size reduces the power consumption difference between the CPU and the GPU*; for larger images (8KFD and FHD) the CPU consumes on average 20% less power than the GPU, and for smaller images (QVGA, VGA and SXGA) even up to 40% less.

c) *the FPGA is a category of its own*; regardless of the filter size, the FPGA consumes about the same amount of power (which is expected considering that it always uses the same number of CLBs).

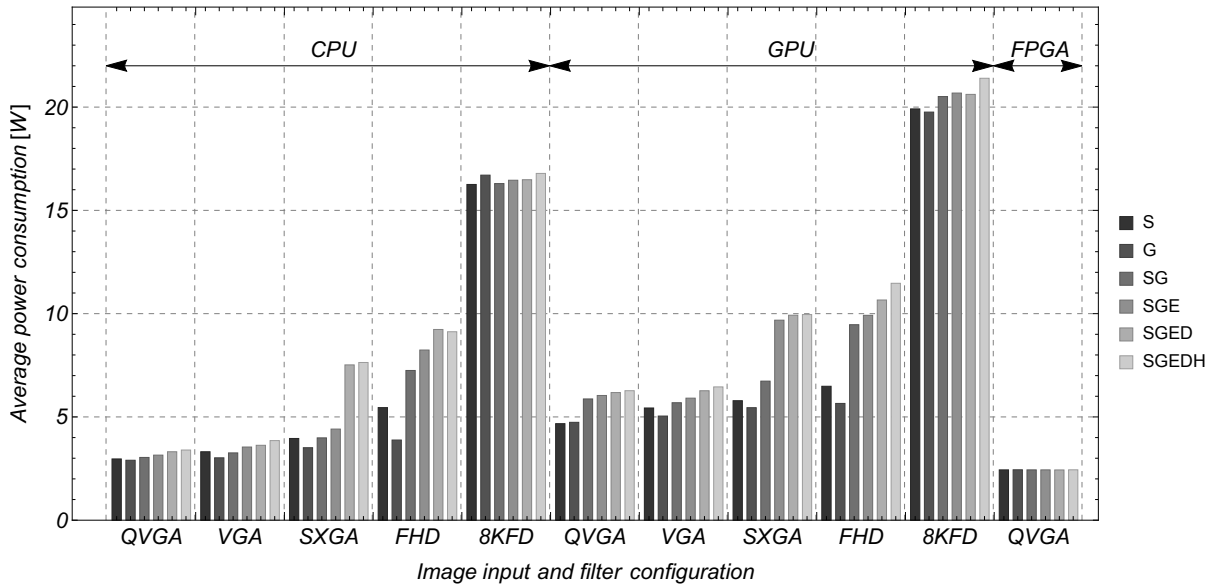

Figure 4.10: Image filtering component, average power consumption in Watts, for different inputs and operations – barchart

Since the *lower is better*, it seems reasonable to assume that the best computing unit

is the FPGA, followed by the CPU and the GPU. However, there is a drawback with the FPGA measurement since the used IP core only supports the QVGA resolution. Regardless of this, from the design principles of the FPGA it is known that the power consumption depends on the amount of used CLBs. Since the current IP always hosts all the filters, but only a selected few are used, it constantly uses the same number of CLB-s, which results with a constant power consumption. Similar observations are made by Flowers [41] and Kestur [57].

Object detection component

Table 4.13 shows the average raise of power consumption expressed in Watts for the Object detection component with different inputs. Considering the VGA row and column II of the CPU, the number 2.5818W is interpreted as the average raise of average power consumption in the case when the CPU is executing the Object detection component, with a second VGA image (previously explained in Table 4.4).

Table 4.13: Object detection component, average power consumption in Watts, for different inputs and operations (I – III different images, with II not containing the object to detect).

	CPU			GPU		
Image	I	II	III	I	II	III
VGA	2.6470	2.5818	2.5791	4.3107	4.2505	4.1807
FHD	8.1050	8.0098	7.8658	17.2423	17.3589	17.3035

The visualization of the Table 4.13 is presented in Figure 4.11. The data is grouped by the platform and the image size. As expected, the CPU is more power efficient than the GPU. For VGA images the CPU, on average, consumed 39% less power than the GPU, while for the FHD image it consumed 54% less. This shows that with increase of the image size, the power consumption of the GPU raises. As a side note, the measurements have shown that using the same Haar classifier for object detection, the CPU was more accurate in detecting an object it was looking for.

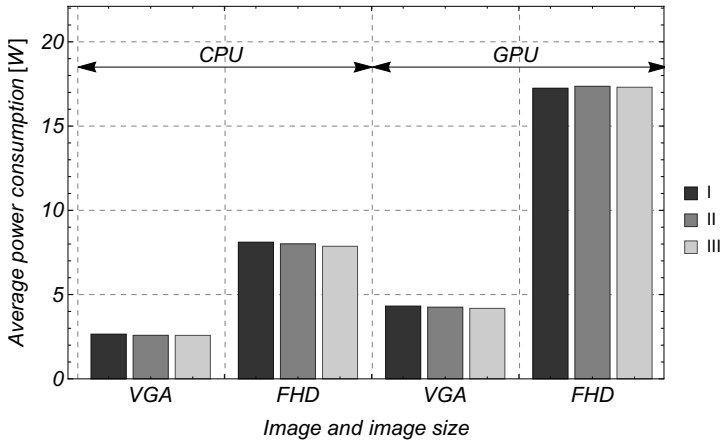

Figure 4.11: Object detection component, average power consumption by a computing unit in Watts – barchart (I – III different images, with II not containing the object to detect).

Remaining components

The results of measuring average power consumption for the Image input, Main controller, Mission manager and Actuator control components are shown in Table 4.14 and in Figure 4.12. Considering the functionality of the remaining components, it is understandable that they can be deployed only on the CPU.
As one can notice, the Image input (II) component has the largest average power consumption (since it uses additional hardware, Microsoft Kinect), followed by the Main controller (MC), the Mission manager (MM) and the Actuator controller (AC).

Table 4.14: Image input, Mission manager, Main controller and Actuator control components, average power consumption in Watts for different operations

	CPU		CPU
Image input		Main controller	
RGB	17.6445	Send commands	1.2998
IR	18.6397	Keyboard enable	5.1148
RGB depth	13.0443	Load image	6.9616
RGB + IR	19.6724	Set image	10.7998
		Initiate FPGA comm.	7.2446
Mission manager		Actuator control	
Load mission	10.7572	Initiate	1.1435
		Send message	3.1826
		List ports	1.0401

Notice that the average power consumption of each of these components differs depending on the called operations. Consider the Main control component and notice that the *Send command* operation increases the average power consumption for only 1.29 W,

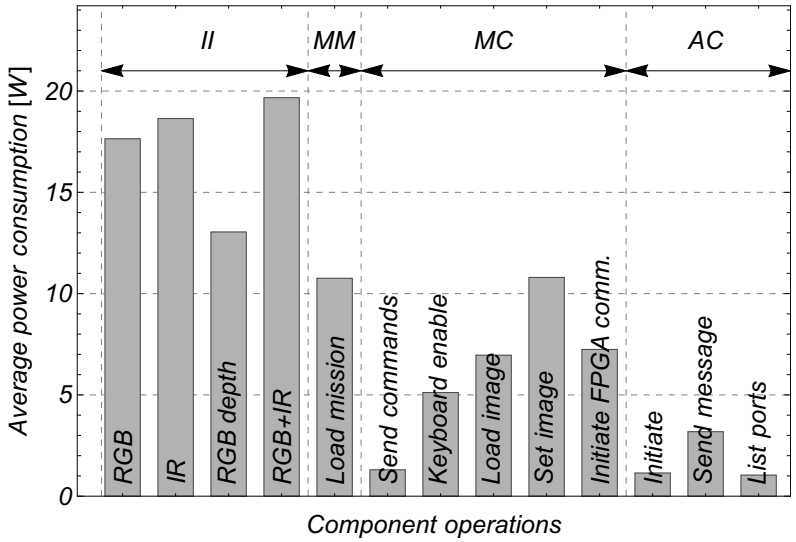

Figure 4.12: Remaining software components, average power consumption by a computing unit in Watts – barchart

while the *Set image* operation increases it for 10.79 W. Therefore it would be very misleading to claim the average power consumption of this component is simply the mean value of all the values presented in the table. So to be exact in making claims about the average power consumption it is necessary to know which operations are called to be able to determine the real average power consumption of a given component. This will be further discussed later on, but first the results for the average execution time will be shown in the next section.

4.3.3 Software component performance results – average execution time

Image filtering component

The Table 4.15 shows the average execution time expressed in milliseconds for each operation of the Image filtering component. The structure of the table is equivalent to the ones used to present the average power consumption of this component.

Table 4.15: Image filtering component, average execution time in milliseconds, for different inputs and operations

	CPU					GPU					FPGA
Resolution	QVGA	VGA	SXGA	FHD	8KFD	QVGA	VGA	SXGA	FHD	8KFD	QVGA
S	1.78	4.29	16.65	26.91	769.28	0.32	1.28	5.06	8.49	113.72	124.19
G	2.29	6.79	25.61	42.32	1253.36	0.60	2.15	8.69	14.41	189.51	124.42
S,G	3.11	8.36	31.74	52.80	1590.83	0.83	3.46	13.67	17.43	289.80	125.10
S,G,E	3.32	10.29	37.55	67.07	1909.40	1.16	5.00	14.38	16.87	380.75	125.47
S,G,E,D	3.63	11.11	49.26	76.24	2235.89	1.59	7.00	15.48	19.39	529.36	124.87
S,G,E,D,H	4.04	11.47	50.04	80.43	2345.60	1.49	5.80	15.12	18.22	481.97	124.62

To put these values in the perspective, the data is visualized in the Figure 4.13. It reveals several interesting details:

a) *increasing the filter size increases the execution time*, however not proportionally, once again there is a positive synergy effect present. The case in which the CPU is processing a FHD image shows this effect very clearly. Having five filtering operations (SGEDH) does not increase the execution time five times over. Moreover, in the GPU–QVGA configuration, notice that there is almost no synergy effect. It is obvious that GPU performs much better when using multiple operations, organized as in a pipeline.

b) *increasing the image size increases the execution time*, which is expected. The unit with the best execution time is the GPU. For larger images (FHD, 8KFD) the CPU takes on average 76% more processing time than the GPU, but for smaller images (QVGA, VGA, SXGA) the difference is a bit less, 63%.

c) *FPGA is once again a category on its own*, regardless of the filter size, the average execution time is always almost the same and also the poorest.

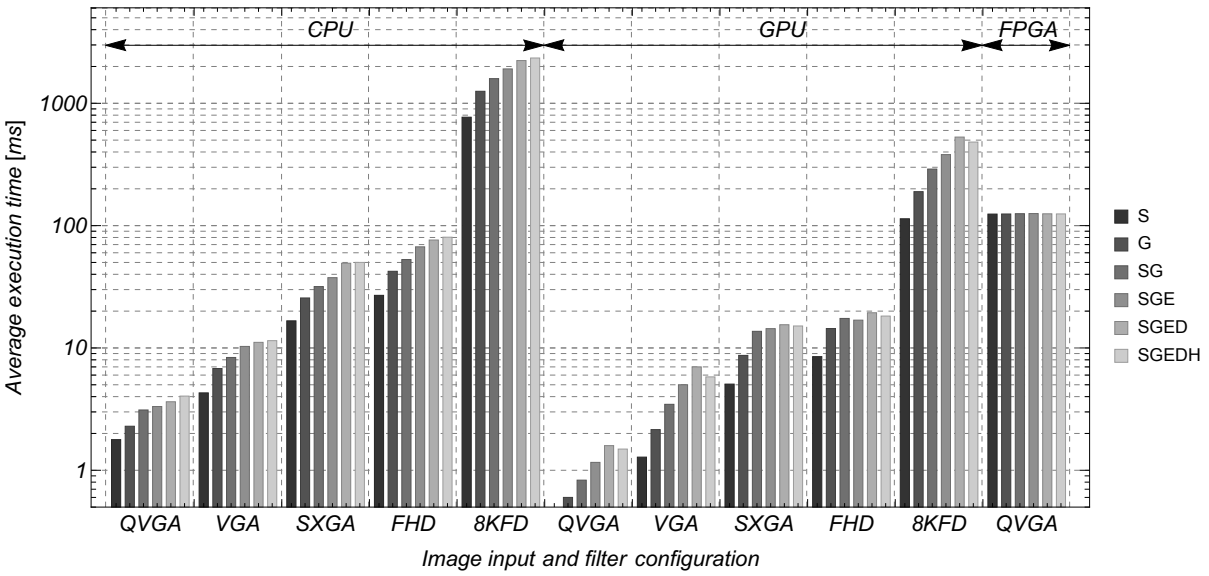

Figure 4.13: Image filtering component, average execution time in milliseconds – barchart (logarithmic scale)

Object detection component

The following table (Table 4.16) shows the average execution time expressed in milliseconds for the Object detection component. For the interpretation example, consider the II column of the CPU and the VGA row. The number 22.98 ms represents the average execution time of the Object detection component with the second VGA input image while allocated to the CPU. The inputs and the table structure is the same as its average power consumption equivalent.

Table 4.16: Object detection component, average execution time in milliseconds, for different inputs and operations (I – III different images, with II not containing the object to detect).

	CPU			GPU		
Image	I	II	III	I	II	III
VGA	22.37	22.98	21.32	5.70	5.72	5.70
FHD	185.78	186.31	184.80	6.26	6.25	6.21

The visualization of Table 4.16 is presented in Figure 4.14. The structure of the diagram is the same as its average power consumption equivalent. One can notice that changing the image size affects the GPU a lot less than the CPU. For the VGA image, on average, the CPU took 74% time more than the GPU, and for FHD image, this increased to 97%.

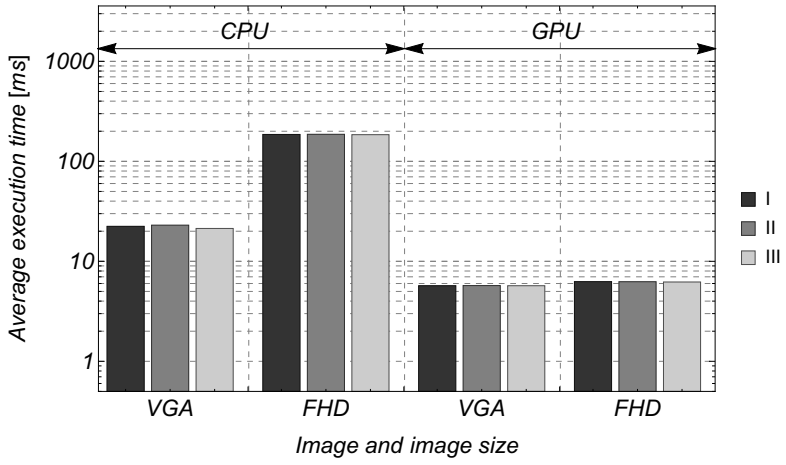

Figure 4.14: Object detection component, average execution time in milliseconds – barchart (logarithmic scale, I – III different images, with II not containing the object to detect))

Remaining components

Finally, the measurement results for the remaining components are shown in Table 4.17. Since some of the values differ in orders of magnitude, the graphical representation shown in Figure 4.15 reveals some interesting details. Depending on the operations provided by each of the components, notice that the Image input component is the most consistent one, i.e. the averages do not deviate as much as for the other

components. The Main controller and the Actuator control components have operations which differ even at the order of magnitude.

Table 4.17: Image input, Mission manager, Main controller and Actuator control components, average execution time in milliseconds for different operations

	CPU		CPU
Image input		Main controller	
RGB	3.27	Send commands	7.15
IR	2.26	Keyboard enable	0.01
RGB depth	2.73	Load image	57.14
RGB + IR	5.15	Set image	1.55
		Initiate FPGA comm.	58.92
Mission manager		Actuator control	
Load mission	0.80	Initiate	0.00216
		Send message	7.21300
		List ports	0.00065

The conclusion is very similar as for the average power consumption; it is nor practical, or correct but rather highly misleading, to use the average execution time of the component as the sum of averages divided by the number of operations.

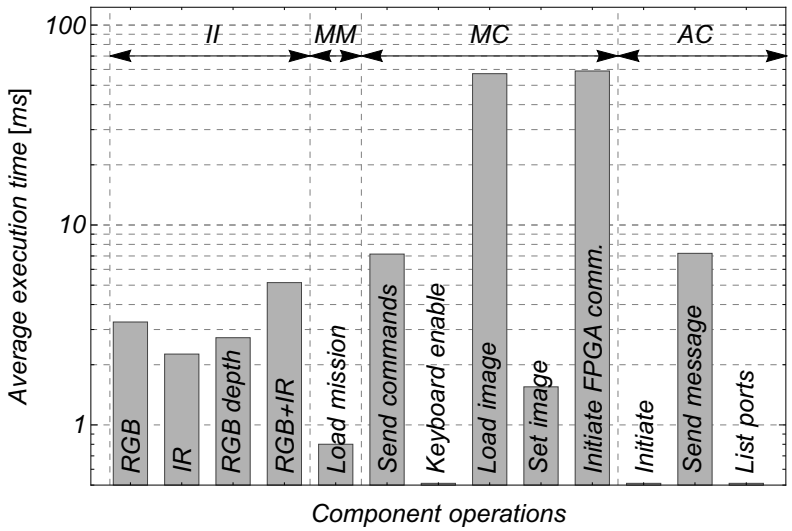

Figure 4.15: Remaining components, average execution time in milliseconds – barchart (logarithmic scale)

Having all the measurement results and considering all the observations made in this section, the next one presents a more detailed discussion about the implications of the collected data.

4.3.4 Discussion

Having the measurement results for both the average power consumption and the average execution time for each operation provided by each of the components, reasoning about the data one could reach the following conclusion:

Firstly, consider the average power consumption. Although the FPGA has the best performance for the Image filtering component, in this particular case it supports only one image resolution. Therefore, the CPU performed the best, while the GPU performed the poorest. For the Image detection component, the CPU also performed better than GPU. Secondly, considering the average execution time, for the Image filtering component the GPU had the best overall execution time, followed by the CPU and the FPGA, which performed surprisingly bad. As for the Object detection component, the GPU performed much better than the CPU. It would be safe to assume that the FPGA and CPU are the best from the power consumption point of view, while the GPU is the best considering the average execution time.

Although it may seem reasonable, that kind of reasoning is seriously misleading. The objective of this work is not to discover or to make a claim which platform is the best, but rather to hypothesize that no single platform is the best. Furthermore, the goal is to find the best software allocation given the heterogeneous computing platform. Therefore, before making any conclusions, there are two points to be considered.

The first one is with the regard to the FPGA. It seems that the FPGA performed very badly considering its the average execution time. However, it must be taken into account that it does include the end–to–end processing time. This time consists of the *data loading time*, the *data processing time*, and the *result returning time*. According to the results, the real execution time (only) took on average 0.08% of the total time. Therefore, the FPGA was actually extremely efficient, however the data transfer cost was quite larger since it was connected to the rest of the system via a LAN cable. Modern FPGAs contain both the CPU and the GPU on a single board, therefore this issue could be easily handled. Another case for the FPGA is the fact that only one IP core was available. There is no doubt that with better access to IP cores, focused on more efficient image processing this issue could be handled and the FPGA could perform just as well (given the way FPGAs work, explained in subsection 4.3.2, average power consumption, point c).). Some research results point out that in some cases it can outperform GPUs [6, 41, 112].

The second point is regarding the CPU and the GPU. The first one seems to be better in power efficiency while the latter seems to be better in processing performance. However, it would be reasonable to assume that the GPU must be better. Regardless to the fact that it consumes more power, it performs the processing much faster. So, overall it should be better (high performance = low power, as suggested by Wolf [121]). To

further clarify this claim, it is necessary to consider the average energy consumption[7] of both computing units.

The comparison of the average energy consumption is presented in the following images. For the Image filtering component, Figure 4.16 shows the percentage difference of average energy consumption between the CPU and the GPU while the Figure 4.17 shows that difference between the CPU and the FPGA, and finally Figure 4.18 shows the percentage difference in energy consumption between the CPU and the GPU for the Object detection component.

The data presented in these images is interpreted as follows. Consider the value of 71% in the Figure 4.16 for the configuration S-QVGA. This number indicates that the GPU's energy consumption was 71% less than the CPU's while using Sobel filter with a QVGA image. As one may notice, all three images are presented as heat–maps to enhance the visibility of the differences. In Figure 4.17 one can notice that the CPU used 98% less energy than the FPGA (with data transfer included). In the Figure 4.18 one can notice that for the Object detection component the GPU used between 59% and 95% less energy than the CPU.

The most interesting observations however, can be made in Figure 4.16. In most cases the GPU uses less energy than the CPU. This can be seen by redder areas. In spite of that, there are cases for which the advantage of the GPU is very low or none, this can be seen in bluer areas. The figure shows a very clear blue cluster focused around QVGA–VGA images with SGE–SGEDH filters. This suggests that when faced with making a design decision, a software architect needs additional considerations to decide where to allocate a component. Considerations like the GPU price, development effort, data transfer costs, etc. Therefore, an architect should weight the benefits and drawbacks of introducing new computing units to the system.

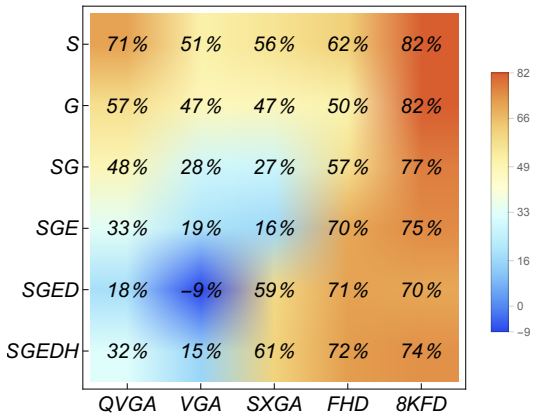

Figure 4.16: Redder means greater percent of CPU energy consumption over GPU. Bluer area is where CPU is comparable to GPU.

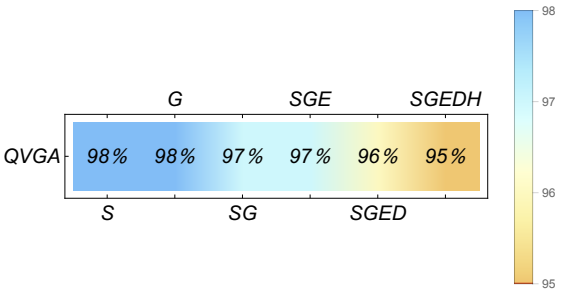

Figure 4.17: Redder means less percent of FPGA energy consumption over CPU. CPU consumes less energy (end–to–end).

[7]Energy consumption is calculated as $E = P \cdot t$

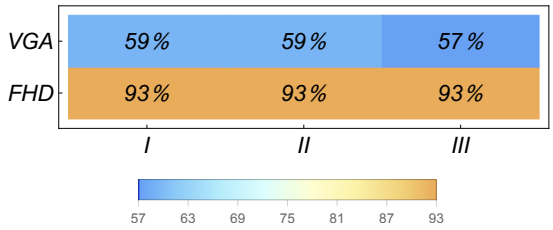

Figure 4.18: Redder means greater percent CPU energy consumption over GPU.

One could suggest that for the SGED-VGA case it is better to use the CPU than the GPU, and rightly so. However, different implementations would produce completely different heat maps and such claims couldn't be made.

The one certainty highlighted by the data is that, a conclusion to decide which platform is the best and what configuration should be used is not simple and straightforward. As the matter of fact the term *best* is very vague in this case. A software architect decides what is the best with consideration to a lot of different parameters ranked by (different) importance, i.e. multiple criteria.

In addition to this, a very important case presented by the measurements is that an architect should also consider different *scenarios*. The scenarios determine the usage of software components, their operations, and the expected input data. If software component usage scenarios are considered in the allocation decision, they can greatly improve it, or when they are left out, make it worse.

The conclusion of this brief discussion to consider different software usage scenarios, and that this work in no way suggests which computing unit is the best, but rather that different implementations can have completely different results. Nonetheless, the underlying decision making process to decide how the software allocation takes place remains the same. In this case, it is argued that the I-IV framework is capable in doing so (**RG-2**).

4.4 Summary

This chapter presented the results of measuring the average power consumption and the average execution time of the software components allocated across different computing units. The use case is built upon the tracked robot TiWO, equipped with the heterogeneous computing platform. This chapter also covers, in detail description of the robots mechanics, hardware (electronics) and software.
There are three computing units, a CPU, a GPU and an FPGA for which detailed specifications given in the hardware related subsection 4.1.1. The software for profiling consists of six components: Image input component, Image filtering component, Mission manager component, Main control component, Object detection component and the Actuator control component.

The data collected by component profiling suggests that with the respect to the average power consumption, the FPGA seems to be the most efficient one, followed by the CPU and the GPU. The GPU is the most power hungry component, however it has shown the best data processing performance with respect to the average execution time. While the FPGA was very efficient in that regard as well, using its end–to–end data revealed that its data transfer time is huge and that it increases the overall performance of the system. Having a SoC platform with all three components on a single board, such communication overhead would certainly produce different results, where these data transfers would probably be reduced.

Regardless of the measurement results, determining the best computing platform proved to be very hard since the term *best* is ambiguous. The main goal is to determine the allocation of software components to utilize the best aspects of all computing units. Also different considerations, i.e. decision parameters can have different importance for different scenarios. Hence, it is suggested to use multiple criteria approach with strong emphasis on considering the software component execution scenarios. These can greatly affect the overall system performance since there is little reason that for one scenario, the optimal software architecture should fit also for other scenarios in which different input data causes components to behave differently, i.e. to exhibit different EFPs. So by taking different execution scenarios into account, a software architect can improve the system's performance.

CHAPTER 5

I-IV FRAMEWORK VERIFICATION

The challenge of this chapter is to verify the I-IV framework by putting the established hypotheses to the test. The verification is performed by proposing six allocations $\alpha^{(1-6)}$ obtained both manually and computationally. The performance of these allocations is predicted by the weighting function $w(\alpha)$ defined by the component allocation model $\mathbb{M}_\alpha$ and evaluated by measurements. For the I-IV framework to be valid, the allocation performance obtained by these methods should match.

5.1 $\mathbb{M}_\alpha$ validation

The assumption of the hypothesis **H-1** is that the component allocation model $\mathbb{M}_\alpha$ correctly represents the system performance if the model substantially represents its extra–functional properties. In order to verify the hypothesis **H-1**, two aspects of the $\mathbb{M}_\alpha$ are considered; a) its trustworthiness and b) its substantial validity.

The trustworthiness of the component allocation model $\mathbb{M}_\alpha$ is verified by comparing the performance of the allocation predicted by the weight function w and the performance of the allocation obtained by measuring its behavior on the real–world platform. This section presents such verification of the model through the following two step experiment.

In the first step a four different allocations will be ranked according to their performance given by the weighting function w of the $\mathbb{M}_\alpha$ model. In the second step, these allocation will be implemented on the previously described TiWo platform. The performance of these allocations will be measured with the same rigorous criteria as presented in chapter 4. Like the performance of allocations in the first step, these will also be ranked in a list. If the rankings obtained by the first and second step match, it can be concluded that the component allocation model $\mathbb{M}_\alpha$ accurately represents and predicts the performance of the allocation for the heterogeneous computing platform. Hence, **H-1** will be accepted. However, if there is any difference between two raking lists, **H-1** will be rejected.

Furthermore, an important aspect of the hypothesis **H-1** is related to the content, i.e. the extra–functional properties represented in the $\mathbb{M}_\alpha$ model. All the data needs to be

substantially valid, meaning that the content needs to be a) of a certain quality and b) of a certain type. The quality (a)) referrers to the statistical *representability* of the input data, as it was described in the previous chapter. All the measurements need to satisfy a statistical confidence interval of 95%.

b) refers to the content type represented in the model. It requires that all the extra–functional properties should be a) quantifiable and b) represented in a less–is-better way, i.e.the lower its quantified value is, the better for the entire system. This is due to the fact that the I-IV framework minimizes the weight w function in the component allocation model $\mathbb{M}_\alpha$.

The Research and Technology Organization (RTO) by NATO refers to non–functional properties as —*ilities*, i.e. a set of properties which describe the behavior of the system rather than its function. Typical examples of these include; reliability, availability, fault tolerance, testability, maintainability, performance, software safety, software security, etc. [79]. All of which can be quantified in a less–is-better way. Some of them are obtained by measuring, while others can be approximated. Needless to say, the ones obtained by measurement work better with the model.

Having that said, according to a Classification Framework for Software Component Models by Crnkovic et.al. [26], the following component modes should be applicable to $\mathbb{M}_\alpha$: BIP, BlueArX, COMDES II, Fractal, Koala, Palladio, PECOS, PIN, ProCom, ROBOCOP, RUBUS, SaveCCM, SOFA 2.0,

Since both average execution time and average power consumption are quantifiable and their value is such that lower–is–better, the inputs to the component allocation model $\mathbb{M}_\alpha$ used in this thesis is substantially valid and can be used to test the hypothesis **H-1**.

5.1.1 Experimental execution scenarios

The measurement results obtained in the previous chapter revealed different input parameters for an allocation can largely impact the overall system performance so a software architect should consider multiple different execution scenarios. Having that said, the experimental allocations for testing the hypothesis **H-1** consider two scenarios which differ by the processing intensity.

Furthermore, to verify the $\mathbb{M}_\alpha$ model it is necessary to have access to several implementations of various execution scenarios. Since there are only a few components available, their granularity will be changed and they will be separated into several smaller components. Currently, the Image filtering and the Object detection components allow for their granularity to be reduced in a way that each of their operations can be repackaged as a single component. The Figure 5.1 shows all the available components after repackaging original ones. In the real–world use cases, a software architect determines the level of granularity.

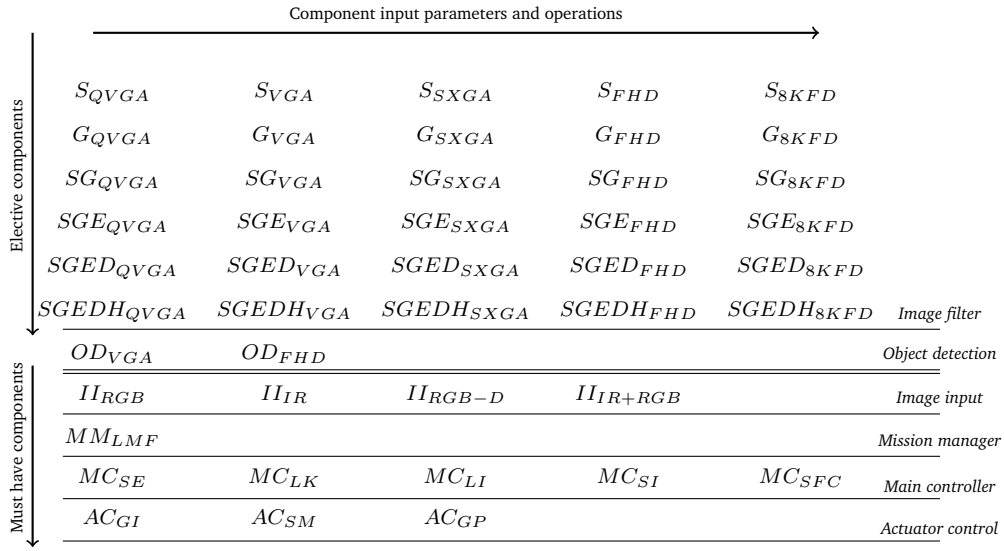

Figure 5.1: The Image filtering and the Object detection components let software architect make a lot of design choices, therefore these are named *elective components*, while the rest of components are obligatory and offer less design choices, these are named *must–have components*.

The new components which emerged after repackaging the Image filtering and the Object detection component are *elective* components, meaning that a software architect gets to choose whether or not to introduce them in the allocation model. The rest of the components are *must–have components*, since all of their operations are necessary for the normal operational flow of the case–study platform, the TiWo robot (subsection 4.1.1). The experimental scenarios include the components in the first two columns shown in Figure 5.1. These 32 components will be organized into scenarios with sequential, i.e. pipelined execution. Since the Image filtering and Object detection components are the part of its vision system, a pipelined architecture is a reasonable choice since each image needs to be processed by some filtering algorithms to enhance edges, remove colors, remove certain layers, etc. This is usually followed by analyzing the processed images and discovering meaningful information for the robot. The order in which the components are be organized in a pipeline is arbitrarily selected.

The selected experimental scenarios used for the verification are the following:

Scenario 1:

$S_{VGA} \rightarrow SGE_{VGA} \rightarrow SGE_{SXGA} \rightarrow SGEDH_{SXGA} \rightarrow OD_{VGA} \rightarrow OD_{FHD} \rightarrow SGED_{FHD} \rightarrow SGEDH_{FHD} \rightarrow S_{QVGA} \rightarrow G_{QVGA} \rightarrow SG_{QVGA}$.

Scenario 2:

$SG_{SXGA} \rightarrow SGE_{SXGA} \rightarrow S_{FHD} \rightarrow G_{FHD} \rightarrow SG_{FHD} \rightarrow SGE_{FHD} \rightarrow SGEDD_{FHD} \rightarrow OD_{FHD} \rightarrow S_{QVGA} \rightarrow SGE_{QVGA} \rightarrow SGEDH_{QVGA}$.

The arrows signify the execution order and as it can be seen by the input image sizes, the Scenario 1 is less intensive for processing, while the Scenario 2 requires more processing

due to larger image inputs.
For both of scenarios three allocations will be generated and ranked according to their performance:

a) a manual allocation, not defined randomly but as a best educated guess based on the results in chapter 4.
b) a solution obtained by the I-IV framework without using the synergy effect approximation.
c) a solution obtained by the I-IV framework with application of the synergy effect approximation.

5.1.2 Manually obtained allocations

Both previously defined scenarios need to be allocated on the TiWo platform. Regarding the allocation choices related to the components defined in the Scenario 1, three[1] components can be allocated on all three computing units (CPU, GPU and FPGA), while the rest of the components can be allocated on only two computing units (CPU, GPU), hence there are $3^3 \cdot 2^8 = 6912$ possible allocations. The manual allocation for the Scenario 1 is shown in Table 5.1.
In order to avoid any intentional misplacement, the results from section 4.3 were used to place the selected components to the most intuitively straightforward computing unit. Hence, the CPU hosts components with the VGA and SXGA input image sizes, for which, according to the measurements it should excel and even outperform the GPU. The GPU hosts all components with FHD image inputs along with the object detection components, while the FPGA hosts components with QVGA input images. The load of all the computing units is somewhat balanced.

Table 5.1: Manual component allocation for the Scenario 1

Computing unit	Components	Hosted components
CPU	S_{VGA}, SGE_{VGA}, SGE_{SXGA}, $SGEDH_{SXGA}$	4
GPU	OD_{VGA}, OD_{FHD}, $SGED_{FHD}$, $SGEDH_{FHD}$	4
FPGA	S_{QVGA}, G_{QVGA}, SG_{QVGA}	3

In the similar way, the components defined in the Scenario 2 are allocated to the computing units on which they should perform the best. The resulting allocation is shown in Table 5.2.
The Scenario 2, as it may be noticed involves much heavier input data than the previous one, and thus most of the components are allocated on the GPU. The component

[1]Filtering components hosted by the FPGA can only be in QVGA (subsection 4.1.2). There are three QVGA components which can be hosted on all computing units, the remaining 8 can be only hosted by the CPU and GPU.

Table 5.2: Manual allocation, Scenario 2

Computing unit	Components	Hosted components
CPU	SG_{SXGA}, SGE_{SXGA}	2
GPU	S_{FHD}, G_{FHD}, SG_{FHD}, SGE_{FHD}, $SGEDH_{FHD}$, OD_{FHD}	6
FPGA	S_{QVGA}, SGE_{QVGA}, $SGEDH_{QVGA}$	3

allocation options are the same as for the Scenario 1, meaning that there are 6912 possible allocations. The load balance of the resulting allocation is skewed toward the GPU since it hosts six components, followed by the FPGA with three components and the CPU with only two components. The reason why the GPU has the highest load is because the measurements have shown that the CPU does not have good performance when it hosts components with FHD image inputs.

5.1.3 Obtaining allocations by *I-IV* framework

The inputs for the $\mathbb{M}_\alpha$ model are collected in chapter 4. Since the focus is on the average power consumption and the average execution time, the communication parameters for the model were set to the default values (all zeros or all ones). As such, they do not have an impact on the resulting allocation since this is not desirable in this particular case.

For both scenarios, the input values are the same for all parameters except for the Computing unit hosting capability matrix ($\mathcal{D}$) and the Resource requirement array ($\mathcal{T}$). For the Scenario 1, these two inputs are indexed with the number 1 and for the Scenario 2, with number 2. The original values (obtained by measurements) of arrays and matrices presented in the following section can also be found in the Table 4.12 and the Table 4.13 for average power consumption, and for the average execution time in the Table 4.15 and the Table 4.16. The next subsection shows the input values for the $\mathbb{M}_\alpha$ model.

I-IV framework input data

The following expression (5.1) respectively presents the Communication intensity matrix $\mathcal{K}$, Platform communication cost $\mathcal{C}$ and Bandwidth matrix $\mathcal{B}$. From these inputs it can be seen that the matrix $\mathcal{K}$ allowed each component to communicate with every other component without limits. Matrix $\mathcal{C}$ shows that the communication between all the components is occurring with the same intensity, while $\mathcal{B}$ shows that there are no bandwidth limitations. These matrices are populated by the default data, and the respective order of components in the matrix is the following: S_{QVGA}, S_{VGA}, G_{QVGA}, SG_{QVGA}, SGE_{VGA}, SGE_{SXGA}, $SGED_{FHD}$, $SGEH_{SXGA}$, $SGEDH_{FHD}$, OD_{VGA},

OD_{FHD}.

$$\mathcal{K} = \begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \end{bmatrix}, \quad \mathcal{C} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}, \mathcal{B} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}, \tag{5.1}$$

The expression 5.2 presents a Resource requirement 3-d array for the Scenario 1. The first dimension in the array represents different software components (11 of them), the second dimension represents computing units (3 of them) while the third dimension represents computing resources (2 of them, i.e. the average execution time and the average power consumption respectively). If the values in the array are compared with data presented in Table 4.13, one may notice different a parameter for the Object detection components ($\mathcal{T}_1$, second column, last two rows). This is because the measurement of the Object detection component was always considered for three different input images, and thus the average time and average power consumption of these three measurements were taken as an input to the $\mathcal{T}_1$.

$$\mathcal{T}_1 = \begin{bmatrix} \begin{bmatrix} 1.78 & 4.29 & 2.29 & 3.11 & 10.29 & 37.55 & 76.24 & 50.04 & 80.43 & 22.22 & 185.63 \\ 0.32 & 1.28 & 0.6 & 0.83 & 5 & 14.38 & 19.39 & 15.12 & 18.22 & 5.7066 & 6.24 \\ 124.19 & 0 & 124.42 & 125.1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \\ \begin{bmatrix} 2.9639 & 3.3125 & 2.9009 & 3.0382 & 3.5405 & 4.4101 & 9.2339 & 7.6358 & 9.1231 & 2.6025 & 7.9935 \\ 4.6727 & 5.4307 & 4.7375 & 5.8686 & 5.9074 & 9.6826 & 10.6593 & 9.9566 & 11.4692 & 4.2473 & 17.3015 \\ 2.4345 & 0 & 2.4395 & 2.4348 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{bmatrix}, \tag{5.2}$$

The next three matrices shown by the expression 5.3 respectively present the Computing unit resource matrix $\mathcal{R}$, the Pairwise resource comparison matrix M_c, and the Computing unit hosting capability matrix $\mathcal{D}$. The values in the matrix $\mathcal{R}$ represent the maximal availability of the resources in question. For the purpose of experiments in this

chapter, all the values are set to the high enough value so that each computing unit can host all the available components. This is because it has been shown that the resources in question cannot be completely consumed any of the computing units, i.e. all the resources have *saturable availability* (chapter 3). Simply put, for the selected scenarios the computing units used here will always have enough resources.
The matrix M_c shows the pairwise comparison of all resources (chapter 3). Respectively, the average execution time, the average power consumption and the communication overhead approximation. The values in the matrix are such that they increase the importance of the average power consumption while the average execution time is of the secondary importance. The communication is setup to be of almost no significance in the allocation decision. Thus, the calculation of the trade–off vector $\mathcal{F}$ resulted with value $(0.36, 0.58, 0.06)$.
Finally, the matrix $\mathcal{D}_1$ makes a constraint by which the FPGA can host only three software components, since 1 means *cannot host*.

$$\mathcal{R} = \begin{bmatrix} 500 & 500 & 500 \\ 500 & 500 & 500 \end{bmatrix}, M_c = \begin{bmatrix} 1 & 0.5 & 9 \\ 2 & 1 & 9 \\ 0.1 & 0.1 & 1 \end{bmatrix}, \mathcal{D}_1 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}. \tag{5.3}$$

Expression 5.4 shows the Mandatory joint component allocation matrix $\mathcal{Y}$ and the Forbidden joint component allocation matrix $\mathcal{X}$. Notice that there are two architectural constraints presented in these matrices. The first one is given by the matrix $\mathcal{Y}$, and it states that components S_{QVGA} and G_{QVGA} must be allocated on the same computing unit, regardless to which. The second constraint, in $\mathcal{X}$ states that components OD_{VGA} and $SGEDH_{FHD}$ must under no circumstance be allocated on the same computing unit. To simplify the allocation decision making, manually obtained allocations do not need to necessarily follow these two constraints.

$$\mathcal{Y} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \mathcal{X} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \tag{5.4}$$

The final 3-d array, presented in expression 5.5 presents how does the resource con-

sumption scale in situations where one computing unit hosts multiple software components. As it was observed by measurements, a component, once it shares resources with other components, changes its requirement positively or negatively, i.e. the aforementioned synergy effect. The expression shows the approximation of that effect. Since this is an approximation, it should be given by measurements or experience, however an architect is not obliged to use it and it can be left out from the model by setting it to the default value (all values in the array should equal to 1). In this experiment, two different allocations will be obtained by the I-IV framework, one with and one without applying the array $\mathcal{S}$. This will also provide the insight about the reliability of a such approximation.

$$\mathcal{S} = \left[\begin{bmatrix} 1 & 0.5 & 0.5 & 0.6 & 0.6 & 0.7 & 0.7 & 0.8 & 0.8 & 1 & 1 \\ 1 & 0.5 & 0.5 & 0.6 & 0.6 & 0.7 & 0.7 & 0.8 & 0.8 & 1 & 1 \\ 1 & 0.5 & 0.5 & 0.6 & 0.6 & 0.7 & 0.7 & 0.8 & 0.8 & 1 & 1 \end{bmatrix} \right. \\ \left. \begin{bmatrix} 1 & 0.5 & 0.5 & 0.6 & 0.6 & 0.7 & 0.7 & 0.8 & 0.8 & 1 & 1 \\ 1 & 0.5 & 0.5 & 0.6 & 0.6 & 0.7 & 0.7 & 0.8 & 0.8 & 1 & 1 \\ 1 & 0.5 & 0.5 & 0.6 & 0.6 & 0.7 & 0.7 & 0.8 & 0.8 & 1 & 1 \end{bmatrix} \right], \tag{5.5}$$

For the Scenario 2, all the inputs remain the same, however since the components changed, the Resource consumption array changes, along with the Computing unit hosting capability matrix. The values of these inputs are respectively shown in expressions 5.6 and 5.7. The order of components is now S_{QVGA}, S_{FHD}, G_{FHD}, SG_{SXGA}, SG_{FHD}, SGE_{QVGA}, SGE_{SXGA}, SGE_{FHD}, $SGEDH_{QVGA}$, $SGEDH_{FHD}$, OD_{FHD}, while the order of computing units remains the same.

$$\mathcal{T}_2 = \left[\begin{bmatrix} 1.78 & 26.91 & 42.31 & 31.74 & 52.8 & 3.32 & 37.55 & 67.07 & 4.04 & 80.43 & 185.63 \\ 0.32 & 8.49 & 14.41 & 13.67 & 17.43 & 1.16 & 14.38 & 16.87 & 1.49 & 18.22 & 6.24 \\ 124.19 & 0 & 0 & 0 & 0 & 125.47 & 0 & 0 & 124.62 & 0 & 0 \end{bmatrix} \right. \\ \left. \begin{bmatrix} 2.9639 & 5.4504 & 3.8794 & 3.9846 & 7.249 & 3.1493 & 4.4101 & 8.2373 & 3.3966 & 9.1231 & 7.9935 \\ 4.6727 & 6.483 & 5.5632 & 6.7314 & 9.4641 & 6.0381 & 9.6826 & 9.9184 & 6.2688 & 11.4692 & 17.3015 \\ 2.4345 & 0 & 0 & 0 & 0 & 2.4352 & 0 & 0 & 2.4383 & 0 & 0 \end{bmatrix} \right]. \tag{5.6}$$

$$\mathcal{D}_2 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \end{bmatrix}, \tag{5.7}$$

The allocations obtained by the I-IV framework

The Table 5.3 presents the allocation for Scenario 1 obtained with the I-IV framework, without using the synergy effect approximation. Eight out of eleven components are allocated to the CPU, only three were allocated to the GPU, while the FPGA was not used at all. The similarity between the manually defined solution can be made since all the components with FHD input images were allocated to the GPU.

Table 5.3: I-IV framework allocation for Scenario 1

Computing unit	Components	Hosted components
CPU	S_{VGA}, SGE_{VGA}, SGE_{SXGA}, $SGEDH_{SXGA}$, OD_{VGA}, S_{QVGA}, G_{QVGA}, SG_{QVGA}	8
GPU	OD_{FHD}, $SGED_{FHD}$, $SGEDH_{FHD}$	3
FPGA	—	0

The result of allocating components in for the Scenario 2 by using I-IV framework without the synergy effect approximation array is presented in the Table 5.4. Seven out of eleven components were placed on the CPU, four to the GPU, while the FPGA was once again left blank.

Table 5.4: I-IV framework allocation for Scenario 2

Computing unit	Components	Hosted components
CPU	SG_{SXGA}, SGE_{SXGA}, G_{FHD}, SG_{FHD}, S_{QVGA}, SGE_{QVGA}, $SGEDH_{QVGA}$	7
GPU	S_{FHD}, SGE_{FHD}, $SGEDH_{FHD}$, OD_{FHD}	4
FPGA	—	0

For the previous two allocations generated by the I-IV framework, the synergy effect array $\mathcal{S}$ was not used. However, by applying it, the I-IV framework results with a different allocation, for the Scenario 1 it is shown in Table 5.5. Six out of eleven components are placed on the CPU, while the remaining five are placed on the GPU leaving FPGA once again unused. It is noticeable that all components with larger image inputs are placed to the GPU.

Applying the array $\mathcal{S}$ in the I-IV framework for the Scenario 2 the resulted with the allocation shown in Table 5.6. For the first time, with I-IV framework, the FPGA is used for two out of eleven components. The CPU hosts five components and therefore, once again the majority, while the GPU hosts four components. GPU hosts only components with FHD input images, while interestingly, the CPU host two such components, but with smaller filter sizes, i.e. less operations are performed.

Table 5.5: Calculated allocation, with synergy tradeoff applied, Scenario 1

Computing unit	Components	Hosted components
CPU	SGE_{VGA}, SGE_{SXGA}, OD_{VGA}, S_{QVGA}, G_{QVGA},SG_{QVGA}	6
GPU	S_{VGA}, OD_{FHD}, $SGEDH_{SXGA}$, $SGED_{FHD}$, $SGEDH_{FHD}$	5
FPGA	—	0

Table 5.6: Calculated allocation, with synergy tradeoff applied, Scenario 2

Computing unit	Components	Hosted components
CPU	SG_{SXGA}, SGE_{SXGA}, G_{FHD}, S_{FHD}, S_{QVGA}	5
GPU	SG_{FHD}, SGE_{FHD}, $SGEDH_{FHD}$, OD_{FHD}	4
FPGA	SGE_{QVGA}, $SGEDH_{QVGA}$	2

Having both manual allocations, and allocations obtained by the I-IV framework the next step is to evaluate them against the weight function w, with an incentive to predict how would they perform on the real–world platform. The result of this step is a raking list of allocations by their performance. This is not obtained manually, but rather with a custom tool; SCALL, the software component allocator.

5.1.4 SCALL – software component allocator tool

A system with n components and m computing units leaves software architects with m^n possible allocations to choose from. This makes finding the best allocation laborious and even infeasible for large n and m. Therefore, a tool which supports and automates this process is essential for aiding in architectural decision making. This section presents a software component allocator, SCALL, a Eclipse plugin used to make allocation decisions.

For the implementation of SCALL several Eclipse based technologies were used; Eclipse Plug–in Development (PDE), Eclipse Modeling Framework (EMF) and Graphical Modeling Project (GMF). This tool chain delivers a good platform for the development of customized tools; EMF provides a modeling framework and code generation facilities, while GMF provides a set of generative components which create an infrastructure for development of graphical editors (as shown in Figure 5.3).
SCALL consists of two main parts; a) *Eclipse based model* editor and b) *PyAllocator*;

Eclipse based model editor – every model represented in SCALL editor corresponds to the metamodel depicted in Figure 5.2. The metamodel uses a standard EMF Ecore notation. Unlike UML or its profiles (e.g. Marte) which could have been also used for SCALL, this metamodel uses a small number of straightforward concepts. Having a

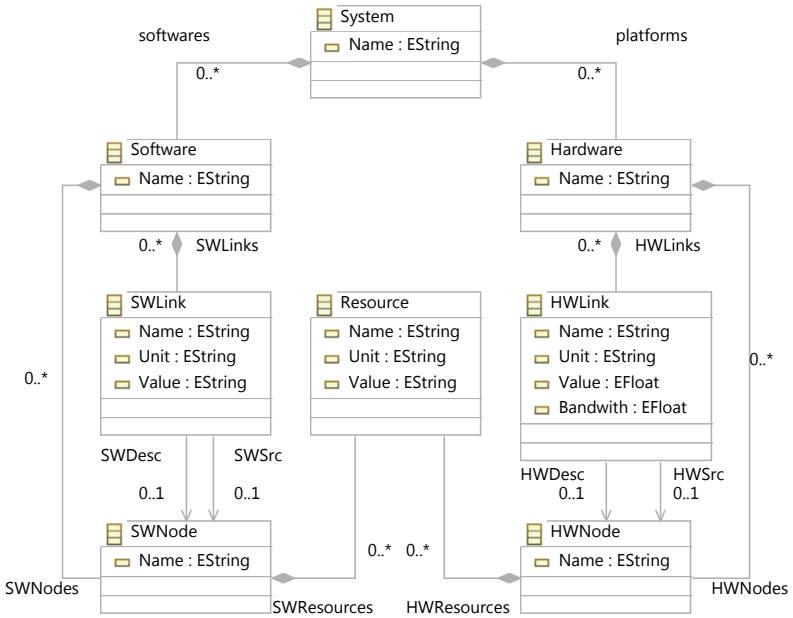

Figure 5.2: SCALL metamodel in Ecore notation

custom model, it is easier to focus on the recognition of important concepts without a burden of large industry standard models and reason about them more effectively. If they prove to be valuable in the future, these concepts can be added to standard models.

Eclipse based model editor is used to design the system. Figure 5.2 shows that the `System` consists of two main compartments: `Software` and `Hardware`. The `Software` is used to host a component model. It can consist of three main concepts; `SWNodes`, `SWLinks` and `Resources`. `SWNodes` represent software components which communicate via `SWLinks` and require some `Resources` to meet their execution specification. A `SWNode` can be associated with any number of links and resources, however all the `SWNodes` need to define values for all resources used in the model.
Similarly, the computing platform is represented by `Hardware` which can host `HWNodes`, `HWLinks` and `Resources`. `HWNodes` represent (heterogeneous) computing units which communicate via `HWLinks` and provide a set of `Resources`. `HWLinks` represent physical communication media for which the current metamodel recognizes two attributes: bandwidth and communication cost (matrices $\mathcal{B}$ and $\mathcal{C}$). `Software` compartment hosts `SWNodes` with corresponding resource demand. For `SWLinks` through which components communicate, allocation model also uses communication intensity (currently a placeholder for future reference, e.g. number of function calls).

PyAllocator – a Python script which employs the multi–objective heuristic allocation method. Python was chosen for its simplicity in handling complex algorithms with rich and fast libraries (e.g. NumPy, SciPy) . From a model created in Eclipse, SCALL extracts previously described matrices which are necessary for making an allocation decision. Once all the matrices are collected, they are converted to JSON format and sent to the

PyAllocator which performs the allocation. Using AHP, PyAllocator calculates weights for each resource (trade–off vector $\mathcal{F}$) and verifies the consistency of the users input. Finally, the solution is obtained by a heuristic algorithm and sent to Eclipse based model editor to be displayed for the user. Detailed description of the decision model can be found in [108].

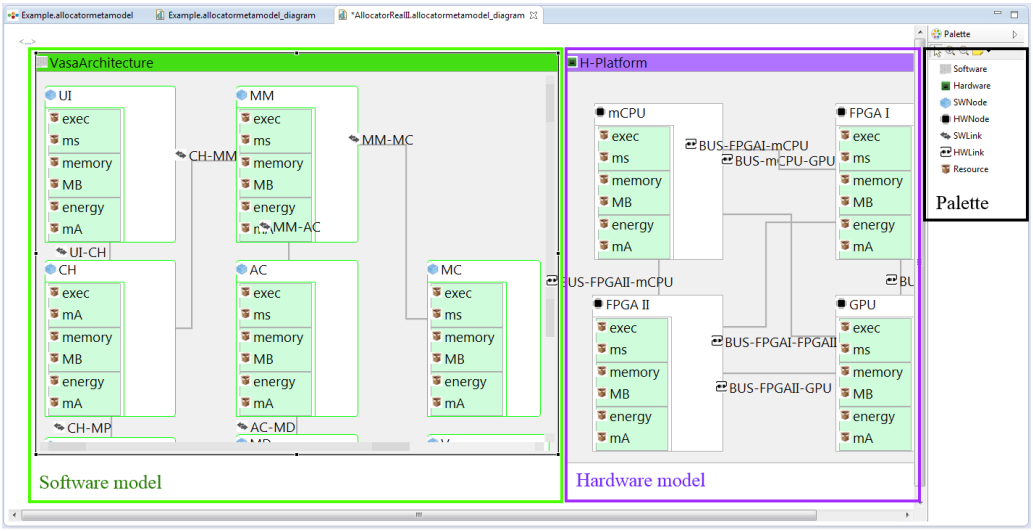

Figure 5.3: SCALL screen shot, showing software (left) and hardware (right) architecture side–by–side

Features

To provide an insight about the software allocation on a heterogeneous platform and also to give a performance estimation, SCALL currently has the following features:

a) *Model creation*, a user can simultaneously model software and hardware architecture of the system. For software components, the user can add resources and model their communication channels. The same can be done for computing units, within the same view.

b) *Model visualization*, enables easier model creation and better overview of the system. Instead of a classical approach of adding elements to a tree-like structure this approach enables drag–n–drop of model elements from a palette and a side–by–side view of both software and hardware architecture.

c) *Pairwise resource comparison*, Having a complete model ready for analysis, the user needs to perform a manual pairwise resource comparison using AHP notation [97] (i.e. determine which resources are more important).

d) *Software component allocation*, SCALL uses the described process to perform a multi–objective allocation of software components on a heterogeneous platform. The procedure results in a (sub–)optimal solution. There are four ways of generating the optimal allocation; a) by a genetic algorithm, b) by simulated annealing, c) by the exhaustive search and d) by randomly generating the allocation.

The full source code is available on `GitHub` for which an URL is given at the end of this thesis.

5.1.5 Experiment simulation – ranking allocations using $\mathbb{M}_\alpha$

Both manual and allocations generated by the I-IV framework are written as an ordered n-tuple, where n represents the number of software components. A value of a tuple at a certain position represents a computing unit to which the software component is allocated. Currently, this value is an element from $\{CPU, GPU, FPGA\}$ in that respective order. The order of the elements in the tuple for the Scenario 1 is the following:

$$\begin{pmatrix} S_{QVGA}, & S_{VGA}, & G_{QVGA}, & SG_{QVGA}, & SGE_{VGA}, & SGE_{SGA}, & SGED_{FHD}, & SGEDH_{SGA}, & SGEDH_{FHD}, & OD_{VGA}, & OD_{FHD} \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \end{pmatrix}.$$

For example, if the fifth element of this arrangement has the value CPU, it means that the fifth component, i.e. SGE_{VGA} is allocated on the CPU. Using this notation the following allocations are weighted and ranked using by the function w from the component allocation model $\mathbb{M}_\alpha$.

The solution obtained by the I-IV framework, with synergy effect array set to default values is:

$$\alpha^{(1)} = (CPU, CPU, CPU, CPU, CPU, CPU, GPU, CPU, GPU, CPU, GPU)\,.$$

The solution obtained by the I-IV framework with applying synergy effect array $\mathcal{S}$ values presented in expression 5.5 is:

$$\alpha^{(2)} = (CPU, GPU, CPU, CPU, CPU, CPU, GPU, GPU, GPU, CPU, GPU)\,.$$

The solution obtained manually, as explained in Subsection 5.1.2 is:

$$\alpha^{(3)} = (FPGA, CPU, FPGA, FPGA, CPU, CPU, GPU, GPU, GPU, GPU, GPU)\,.$$

For the Scenario 2, the values of the solution vector components are the same as for the Scenario 1, however the order of the components changed to:

$$\begin{pmatrix} S_{QVGA} & S_{FHD} & G_{FHD} & SG_{SGA} & SG_{FHD} & SGE_{QVGA} & SGE_{SGA} & SGE_{FHD} & SGEDH_{QVGA} & SGEDH_{FHD} & OD_{FHD} \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \end{pmatrix}.$$

The solution obtained by the I-IV framework, with synergy effect array set to default values is:

$$\alpha^{(4)} = (CPU, GPU, CPU, CPU, CPU, CPU, CPU, GPU, CPU, GPU, GPU)\,.$$

The solution obtained by the I-IV framework with applying synergy effect array $\mathcal{S}$ values presented in expression 5.5 is:

$$\alpha^{(5)} = (CPU, GPU, CPU, CPU, GPU, CPU, CPU, GPU, CPU, GPU, GPU)\,.$$

The solution obtained manually, as explained in the subsection 5.1.2 is:

$$\alpha^{(6)} = (FPGA, GPU, GPU, CPU, GPU, FPGA, CPU, GPU, FPGA, GPU, GPU)\,.$$

The next step is to rank allocations $\alpha^{(1-6)}$. This is performed by evaluating these allocations by the weight function w. The Table 5.7 presents the result of evaluating allocations which do not apply the $\mathcal{S}$ array. The column *Rank* shows the rank of the solution while the column *Evaluation result* shows the value given by the function w. The lower the rank the better the allocation. The Table 5.8 shows the evaluation scores and ranking of the allocations which use the $\mathcal{S}$ array.

Table 5.7: Ranking of allocations $\alpha^{(1)}$, $\alpha^{(3)}$, $\alpha^{(4)}$, $\alpha^{(6)}$ – solutions without using array $\mathcal{S}$.

	Allocation	Evaluation result	Rank
Scenario 1	$\alpha^{(3)}$	3,3888	3
	$\alpha^{(1)}$	2,6957	1
Scenario 2	$\alpha^{(6)}$	3,5908	4
	$\alpha^{(4)}$	2,9374	2

Table 5.8: Ranking of allocations $\alpha^{(2)}$, $\alpha^{(3)}$, $\alpha^{(5)}$, $\alpha^{(6)}$ – solutions using array $\mathcal{S}$.

	Allocation	Evaluation result	Rank
Scenario 1	$\alpha^{(3)}$	1,9348	3
	$\alpha^{(2)}$	1,7155	1
Scenario 2	$\alpha^{(6)}$	2,2326	4
	$\alpha^{(5)}$	1,8626	2

The next step is to implement these software components, allocate them on the TiWo platform and by measurement obtain their real world performance to rank them.

5.1.6 The experiment: manual allocations vs. I-IV obtained allocations

By applying the same measurement procedure and parameters as described in chapter 4, allocations $\alpha^{(1-6)}$ were subjected to the real–world performance evaluation. For each allocation, the average execution time and the average power consumption were measured. The results are shown in Tables 5.9 and 5.10. The weights for the resources shown in the table are obtained by the pairwise comparison (expression 5.3) inspired by the AHP which results with the trade–off vector $\mathcal{F}$. These are the same as for the I-IV framework. The final score (column *Result*) is obtained in the same way as the weight function w evaluates the solution in the model $\mathbb{M}_\alpha$, i.e. as $P_{AVG} \cdot P_W + t_{AVG} \cdot t_W$. With this, the measurement units are dismissed and the result is interpreted as *less is better*.

Table 5.9: Manual allocations vs. allocations generated by the I-IV framework without applying of the synergy effect array

	Allocation	Average power consumption [W](P_{AVG})	Average execution time [ms](t_{AVG})	Time weight (t_W)	Power weight (P_W)	Result	Rank
Scenario 1	$\alpha^{(3)}$	31,0731	8887,6262	0,36	0,58	3217,5678	3
	$\alpha^{(1)}$	32,3924	4153,5888	0,36	0,58	1514,0796	1
Scenario 2	$\alpha^{(6)}$	31,9835	11071,8959	0,36	0,58	4004,4329	4
	$\alpha^{(4)}$	32,1609	6221,8502	0,36	0,58	2258,5194	2

Table 5.10: Manual allocations vs. allocations generated by the I-IV framework with application of the synergy effect array

	Allocation	Average power consumption [W](P_{AVG})	Average execution time [ms](t_{AVG})	Time weight (t_W)	Power weight (P_W)	Result	Rank
Scenario 1	$\alpha^{(3)}$	31,0731	8887,6262	0,36	0,58	3217,5678	3
	$\alpha^{(2)}$	32,9684	4051,0150	0,36	0,58	1477,4871	1
Scenario 2	$\alpha^{(6)}$	31,9835	11071,8959	0,36	0,58	4004,4329	4
	$\alpha^{(5)}$	33,2779	6226,1707	0,36	0,58	2260,7226	2

The most important column is the rank, which for both tables unequivocally match to the ones predicted by the I-IV framework in the previous subsection (Tables 5.7, 5.8). Therefore, **the hypothesis H-1 is accepted** it is henceforth accepted that the *I-IV framework correctly represents the system performance of the heterogeneous computing platform at hand*.

While the aforementioned tables match, it is also interesting to notice that the allocation which applies the synergy effect array performed better ($w(\alpha^{(2)}) = 1447.48$) then the allocation without it ($w(\alpha^{(1)}) = 1514.07$). Since this was also predicted by the model (2.6957 vs 1.7155), such result gives an additional straight to the validity of I-IV framework. However, since this array is obtained by experience, i.e. it introduces a

degree of uncertainty due to its approximation. It is strongly suggested to use it with caution and this in mind.

Experiment details and statistical analysis of the results

The measurements were conducted using the same parameters and method as shown in chapter 4. The mean values, errors and the upper and lower bound of the 95% confidence interval are shown in the table Table 5.11.

Table 5.11: Time and current measurement results used for ranking (direct Minitab outputs)

Allocation	N	Mean (A, ms)	St.dev.	SE Mean	CI-lower (95%)	CI-upper (95%)
Current (A)						
$\alpha^{(1)}$	30	2,7898	0,0333	0,0061	2,7774	2,8022
$\alpha^{(2)}$	30	2,8394	0,0786	0,0146	2,8095	2,8693
$\alpha^{(3)}$	30	2,6762	0,0405	0,0074	2,6611	2,6913
$\alpha^{(4)}$	30	2,7699	0,0796	0,0145	2,7401	2,7996
$\alpha^{(5)}$	30	2,8661	0,1115	0,0204	2,8244	2,9077
$\alpha^{(6)}$	30	2,7546	0,0562	0,0103	2,7336	2,7756
Time (ms)						
$\alpha^{(1)}$	30	4153,59	38,41	7,01	4139,25	4167,93
$\alpha^{(2)}$	30	4051,02	38,39	7,13	4036,41	4065,62
$\alpha^{(3)}$	30	8887,60	167,90	30,70	8824,90	8950,30
$\alpha^{(4)}$	30	6221,85	34,2	6,24	6209,08	6234,62
$\alpha^{(5)}$	30	6226,17	35,78	6,53	6212,81	6239,53
$\alpha^{(6)}$	30	11071,90	131,80	24,10	11022,70	11121,10

The measurements were repeated 30 times to ensure a large enough sample. Notice that the standard deviation is very low for all current measurement data meaning that the true measured value lies in a very narrow confidence interval, for which the upper and lower bound are within two decimal places. Measuring the time produced a higher standard deviation, but since it is measured in milliseconds, the presented values are acceptable since the confidence interval is very narrow.

5.2 I-IV framework performance for large search spaces

The hypothesis **H-2** states that in the case of a large number of allocation choices the I-IV framework finds an allocation which performs significantly better than a set of random allocations. Given a set of components n and a set of computing units m, the search space required to scan in order to find the best allocation is m^n. Since there are no known algorithms that could find an optimal allocation in the polynomial time, a sub–optimal good enough solution is also acceptable. Therefore, this hypothesis will be accepted if following conditions are met:

a) the sub–optimal allocation provided by the I-IV framework is not significantly worse than the optimal allocation obtained by the exhaustive search, i.e. the optimal solution. Minimal sacrifice of the resulting allocation performance for the maximal decrease of the search time is acceptable.
b) the sub–optimal allocation provided by the I-IV framework performs significantly better in the significant number of cases in comparison with a set of randomly generated allocations.

The evaluation of these conditions starts by comparing the performance difference between the optimal allocation obtained by the exhaustive search method and sub–optimal allocation obtained by heuristic methods implemented in the I-IV framework.

5.2.1 Optimization methods

The specification of the I-IV framework does not formally define any method for optimizing the software component allocation, and as such it is the subject to deeper investigation of this subject in the future, however its current implementation in SCALL provides two heuristic methods for obtaining the allocation; by the genetic algorithm and by the simulated annealing algorithm. This subsection presents the comparison of the performance of allocations obtained by four different methods:

1) by using the genetic algorithm (GA),
2) by using the simulated annealing algorithm (SA),
3) by performing an exhaustive search (ES), which finds the real optimal allocation,
4) and finally by generating 30 consecutive randomly generated allocations and considering the best (min), the worst (max) and the average (avg) performing allocation in this set (RAND).

Genetic algorithm implementation and its setup

Genetic algorithm is an adaptive heuristic local search algorithm inspired by the principles of natural selection. It falls into the category of evolutionary algorithms which provide a efficient techniques for addressing large search spaces with a goal of finding the most robust and best rated solution according to a fitness criteria. Mimicking the

same basic principles as observed in nature, the outcome of the genetic algorithm is a an array which represents a sub–optimal, good enough solution to an optimization problem. The algorithm starts by randomly generating the initial population of candidate solutions, or in this case, allocations. Next, it uses tree operators; *the selection operator*, *the crossover operator* and *the mutation operator*. The selection operator evaluates the *goodness* of the solution, i.e. allocation performance by applying the fitness function, which is in this case function w. Then, on the individuals which passed the selection, a crossover operator is applied which produces an offspring carrying the characteristics of both individuals which produced it. This process is also refereed to as mating. Finally, the mutation operator is applied to the offspring, which with some low probability enables them to develop their own new characteristics, or more precisely to change value at the certain position of the array. This is repeated on the individuals which passed the selection criteria and one such iteration is called a generation.

Given the initial values of these parameters one should be careful. If the population size is to loo small the search space will not get good coverage. If the population is to big, the epoch (all iterations) time is increasing and a chance for each individual to explore the neighborhood is somewhat restricted. For the mutation rate, if the value is to high there is a chance to skip over the optimal solution, and if it is to low there is a chance for it to get stuck in the local minimum. As for the crossover operator, for this particular problem it is important that it respects the order of individual chromosomes, i.e. array elements.

Table 5.12: Genetic algorithm - DEAP parameters

Parameter	Value	Interpretation
crossover	cwTwoPoint	executes a two-point crossover on the input sequence individuals
mutation	mutFlipBit	flips the value of the input sequence individual with a certain probability, set to 20% (initial configuration is set to 5% – toolbox parameters)
selection	selTournament	selects k individuals form the input and uses k tournaments of turnsize individuals. The turnsize is set to 3, while k is 300
method	eaSimple	simplest form of the genetic algorithm
population	300	random solutions which are evolved toward better solutions
generations	40	number of iterations

The entire I-IV framework is implemented in Python 3.5 as a part of PyAllocator, and to implement the solver which uses a genetic algorithm, DEAP library was used [40]. It is an evolutionary computation framework for rapid prototyping and testing with support of parallelization and multiprocessing instruments. Table 5.12 presents the setup parameters for the DEAP library.

Simulated annealing implementation and setup

Simulated annealing is an optimization algorithm inspired by metallurgic process of metal annealing. When a liquid metal is cooled rapidly, the molecules within it tend to obtain a more random structure, or what is in metallurgy known as reduced crystallinity. However, if the metal is gradually cooled the resulting material ends up in a state with low tension energy. This basic principle is transferred to the function optimization. The algorithm has a starting and an ending temperature, while each temperature point has an energy value determined by the objective function (in this case function w). Each iteration reduces the temperature for a predetermined value. The temperature is used to control the variability of the candidate solutions, where larger temperatures allow more variations and lower temperatures allow less variations. Unlike other methods, Simulated annealing is prone to resist getting stuck at a local minimum since in the each iteration, the solution is accepted even if it is not the best one. It called a candidate solution which is accepted with a certain probability of being the optimal solution. The search for the best allocation using Simulated annealing is also implemented in Python 3.5 with Simanneal library [2] as a part of PyAllocator. The algorithm parameters are shown in the following table (Table 5.13).

Table 5.13: Simulated annealing - Simannel parameters

Parameter	Value	Interpretation
T_{max}	1000000	Starting temperature
T_{min}	0.5	Ending temperature
i	20000	Number of iterations

As it can be seen in Table 5.13, the initial temperature is very high to allow for high variability of the candidate solutions. However, with gradual cooling, variability is reduced. The Simanneal implementation of the simulated annealing algorithm does not require one to define the change of the temperature in each step, but it does allow to specify the number of iterations. Then, it calculates the temperature change of each step using a nonlinear function.

In this work, these optimization algorithms are not the main research subject so their in–detail analysis was not performed since it is out of scope. There was minor tweaking of input parameters until the algorithms started to produce satisfying solutions. Once that happened the parameters remained the same throughout the research. But still, producing a better optimization method used for allocating software components in a heterogeneous computing environment might be an interesting subject for the optimization community and the future work.

[2] https://github.com/perrygeo/simanneal, accessed in December, 2015

5.2.2 Optimizing the software architecture

The goal of the experiments presented in this section is to evaluate how do the sub–optimal allocations obtained by the heuristic methods compare to the optimal allocations obtained by the exhaustive search, both in the means of time to obtain the solution and the solution precision.

Due to the lack of access to the real–world heterogeneous computing platforms with larger number of software components and computing units, the inputs for the I-IV framework were randomly generated[3]. In particular, the resource availability matrix was generated so that it always provides enough resources, and the inputs related to architectural constraints were generated with a function which is prone to generating more zeros than ones. This means that the resulting platform had less architectural constraints due to the danger of generating systems with infeasible solutions. In this experiment, these are not considered since they would not provide any meaningful information.

Once the inputs were generated, using the I-IV framework the best allocations for each of the generated heterogeneous platforms were optimized, that is, its weight function w was minimized using the previously defined methods (1–4), i.e. GA, SA, ES and RAND.

There there were two experiments performed, the first one dealing with the precision of the optimization methods, and the second one dealing with extreme number of allocation choices.

Determining the best optimization method – experiment 1

The first experiment deals with allocation performance. The goal is to determine how good of a performance have the allocations allocations provided by the heuristic approach in comparison to the real optimal allocations, and a set of randomly generated allocations. Since the search space grows rapidly, only 14 different heterogeneous platforms were randomly generated to experiment with. The ES optimal allocation was calculated only once, while the GA, SA and the RAND allocations were recalculated 30 times.

The results have shown that the GA and the SA always converge in the same allocation and therefore it is unnecessary to repeat them multiple times for the future experiments, if their initial conditions do not change. For the RAND approach, the 30 repetitions generated a variety of allocations with different performances. The performance of allocations obtained by each of the method is shown in Table 5.14 as the average deviation from the optimal allocation in percent.

The columns m, n and k in the table Table 5.14 are respectfully the number of com-

[3]The simulation of heterogeneous platform is implemented with Python 3.5 and executed on a server with 2× Intel®Xeon®CPU E7-4830, 8GB Ram, and Linux

puting units, the number of software components an the number of different resources. The column $t[s]$ shows the average search time to necessary obtain the solution.

Given the search space size ranging between 2^8 and 5^{12} the performance for different methods varies. Obtaining the randomly generated allocations was instantaneous, since the time necessary to generate it is the time necessary to generate a random array of a certain length and values. For the exhaustive search, the time was better than both GA and SA up until 3^7 beyond which it increased exponentially. For the largest search space of of 5^{12} it took more than 1 day and 14 hours to find the optimal solution, while the GA and the SA took significantly less. Also notice that the GA almost regardless of the search space size took about the same amount of time, and for the SA took more time for smaller search space and less time for larger search space.

Table 5.14: Comparison of allocation performance between the optimal solutions, randomly generated solutions and solutions obtained by the genetic algorithm and the simulated annealing.

			ES		GA		SA		RAND			
m	n	k	$t[s]$	dif.[%]	$t[s]$	dif.[%]	$t[s]$	dif.[%]	$t[s]$	dif.$_{min}$[%]	dif.$_{max}$[%]	dif.$_{avg}$[%]
2	8	5	0,179	0	7,839	0%	18,005	0%	0,001	10%	101%	33%
2	9	5	0,475	0	7,922	0%	19,228	0%	0,001	5%	108%	33%
2	10	5	0,819	0	8,556	0%	19,899	0%	0,001	3%	62%	25%
3	7	5	1,333	0	7,537	0%	17,484	0%	0,001	8%	266%	114%
3	8	5	4,000	0	7,654	0%	18,316	0%	0,001	5%	256%	78%
3	9	5	14,000	0	8,246	0%	19,960	0%	0,001	11%	118%	54%
4	8	5	57,000	0	7,985	0%	19,726	0%	0,001	58%	370%	172%
4	9	5	238,000	0	8,602	2%	20,658	2%	0,001	32%	260%	120%
4	10	5	679,000	0	4,277	2%	10,726	4%	0,001	22%	167%	82%
5	9	5	1727,000	0	7,453	6%	18,240	7%	0,001	43%	339%	145%
5	10	5	4298,000	0	4,292	3%	10,681	4%	0,001	108%	355%	195%
5	11	5	24328,000	0	4,400	7%	11,422	13%	0,001	40%	205%	125%
5	12	5	139988,000	0	5,149	5%	13,373	12%	0,001	46%	213%	125%

The column dif.$[\%]$ shows the difference in percentage between allocation performance for the GA, SA and RAND in comparison to the performance of optimal allocation obtained by the ES. For all allocations obtained by the ES this is obviously 0. For the GA generated allocations, the greatest difference is 7%, and for the SA generated allocations it is 13%. Comparing these to the allocations obtained by RAND, the allocations are always worse than allocations from both the GA and the SA. The performance of each generated RAND allocation was also evaluated using the function w, and the table presents the minimal, maximal and average performing randomly generated allocation. The minimal offset is 3% and it occurs in lower search spaces while the largest offset was 370%. On average, the smallest difference (min) was 33% worse than the optimal one, while for the largest (max) this was 195%. In all 14 cases both the GA and the SA provided better allocations. For better overview, the data presented in Table 5.14 is

shown in Figure 5.4.

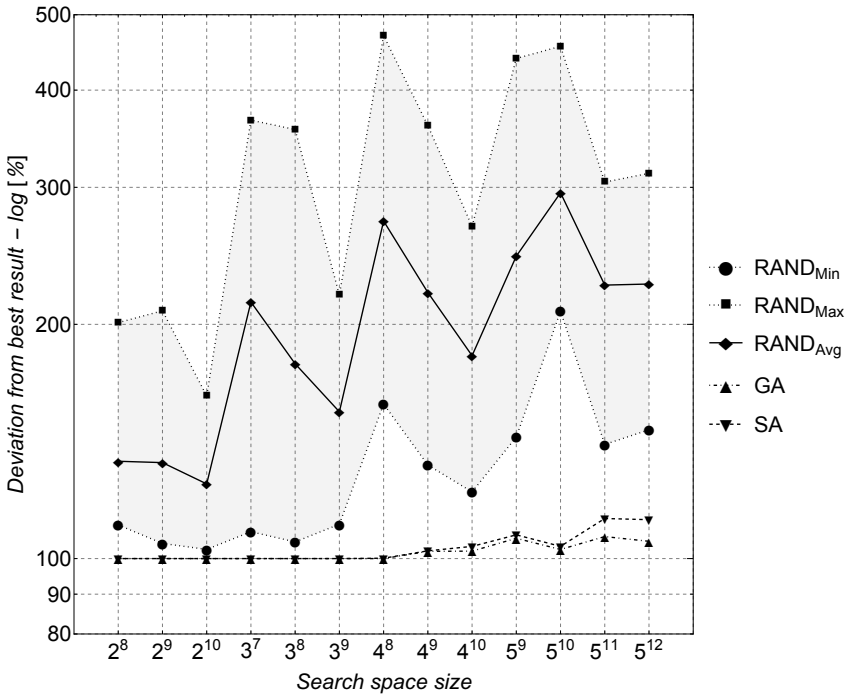

Figure 5.4: Difference in allocation performance in percentage for GA, SA and RAND (min, max, avg.) – logarithmic scale is used.

The gray area shown in Figure 5.4 represents the area in which lies the performance of all allocations obtained by RAND. The performance of the GA and the SA generated allocations are the same as the optimal allocation up to the search space of size 4^8, but toward the end, for the large search spaces, the allocation provided by the GA performed slightly better.

A different perspective of this data is shown in figure Figure 5.5, which presents the performance each methods' optimal allocation by the weight obtained from the function w. The image reveals that the allocation with minimal weight generated by the RAND gets fairly close to the optimal solution in the search spaces between 2^9 and 3^7. But for larger search spaces, the difference starts to grow and never (for these search spaces) reaches it anymore. Also, the image shows the big differences between the best, the worst and the average performing allocation provided by the RAND.

The time necessary to obtain the best allocation by each of the methods reveals interesting information shown in Figure 5.6, which might otherwise be hard to notice in the previously presented Table 5.14. The image uses logarithmic scale because there is a great discrepancy between the time necessary to obtain allocations by the ES and time necessary to obtain allocations by the GA and the SA. The RAND was not considered in this image at this point, since it does hardly depend on the search space size. It can be seen that after 3^9 the GA and the SA significantly outperform the time necessary to obtain the solution by the ES. The time keeps increasing to the point beyond which it is unreasonable to keep waiting for the algorithm to finish. For the last configuration

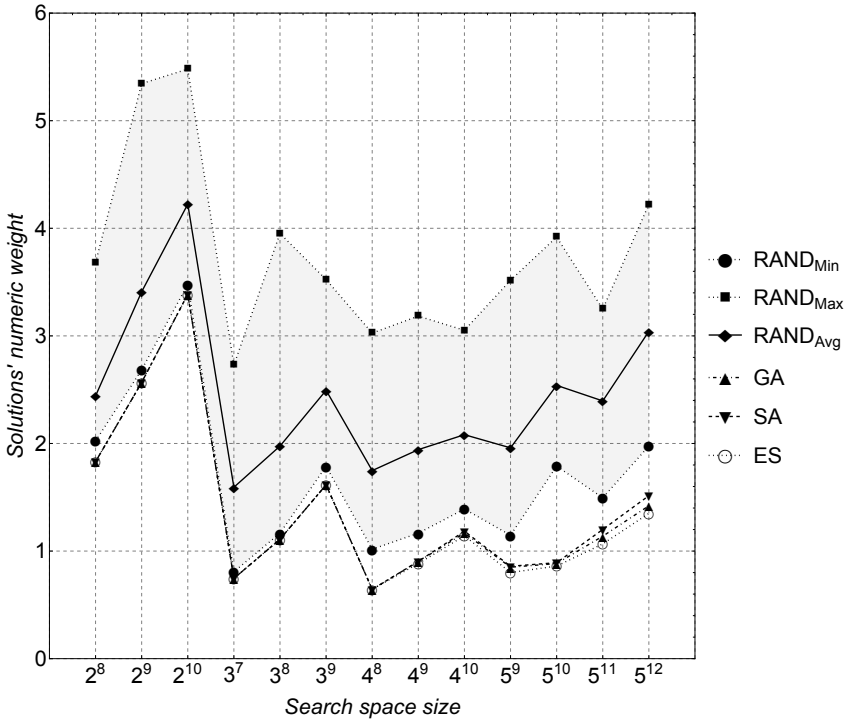

Figure 5.5: Solution weight for ES, GA, SA and RAND (min, max, avg.).

the search space was 5^{12} so for the ES to find the optimal allocation it took more than 1 day and 14 hours, while the GA and the SA it took only about 10 seconds. And the sub–optimal allocations were performing only 5 – 13% worse than the optimal one.

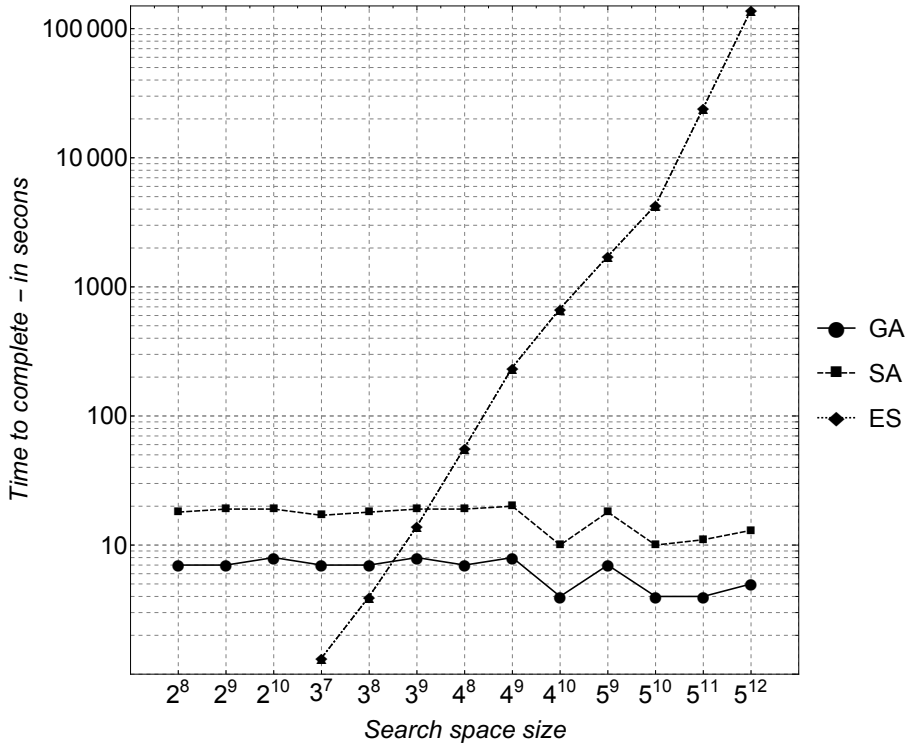

Figure 5.6: The comparison of time necessary to generate the allocation for the ES, GA, and SA approach. Since the scale is logarithmic, values smaller than 1 second are not shown.

Given the results presented in Table 5.14, one can safely sacrifice the allocation performance for the time necessary to obtain it, since the performance of the sub–optimal solution does not differ much from the performance of the optimal allocation. With this, the resulting conclusion of the first experiment is that the GA and the SA, i.e. default optimization methods of the I-IV allocation framework provide a satisfactory sub–optimal allocation.

For the second experiment it is necessary to verify whether the allocations obtained by the standard I-IV allocation framework optimization methods produce significantly better performing allocations than the ones obtained by the RAND method for extremely large search spaces.

Extreme number of allocation choices – experiment 2

The following challenge is to determine the difference between solution performance obtained by the I-IV frameworks GA and SA approach and the RAND approach described previously, but for much larger search space which is beyond the exhaustive search method. In the previous experiment for the largest search space of 5^{12} it took 1 day and 14 hours, but to find an optimal allocation in any search space larger than a dedicated supercomputer would be required, which could maximally exploit the multiprocessing options of the programming language, the operating system and the computing platform. This is not the point, however, the goal is to find the sub–optimal allocation with satisfactory performance in the reasonable amount of time which is also better than (any) randomly generated allocation, which might not always be the case [37].

The following experiment utilizes the GA and SA approaches currently implemented in the I-IV framework PyAllocator and compares the performance of their allocations against the allocations provided by the RAND approach. The heterogeneous computing platform were randomly generated 55 different configurations, for search spaces ranging between 10^{20} and 30^{70}. This represents systems with at least 10 computing units and 20 components and at most 30 computing units and 70 components. Needles to say, these search spaces are huge.

The Table 5.15 shows the result of the experiment. For each input configuration, the GA and the SA allocations were obtained only once (since they always converge to the same solution), while randomly generating allocations was repeated 30 times for which the best, the worst and the average performing ones were considered.

Table 5.15: Comparison between allocation performance between the randomly generated solution and solutions obtained by the genetic algorithm and simulated annealing.

		GA		SA		RAND			
m	n	$t[s]$	w	$t[s]$	w	$t[s]$	w_{min}	w_{max}	w_{avg}
10	20	26,628	2,912	74,593	2,957	0,001	3,808	7,009	5,186
10	21	28,164	2,760	72,611	3,079	0,001	3,953	6,582	5,148
10	22	29,164	2,906	79,888	3,282	0,001	5,363	6,718	6,111
10	23	30,762	3,227	83,862	3,370	0,001	4,070	6,842	5,641
10	24	31,431	3,782	84,711	4,243	0,001	5,576	8,152	6,927
10	25	33,434	4,053	90,629	4,432	0,001	6,456	8,649	7,268
10	26	35,462	4,196	95,492	4,723	0,001	5,782	8,723	6,910
10	27	35,998	4,810	101,252	5,070	0,001	6,356	8,888	7,718
10	28	44,770	5,275	127,298	5,712	0,001	6,851	12,257	9,138
10	29	95,885	5,741	269,554	6,432	0,001	7,266	10,123	9,011
10	30	48,036	6,015	127,904	6,440	0,001	7,730	11,884	9,768
15	30	45,746	5,223	125,520	5,630	0,001	6,262	10,329	8,834
15	31	46,798	5,631	128,692	6,135	0,001	6,392	10,928	9,281
15	32	48,929	5,447	135,545	6,073	0,001	7,401	11,336	9,409
15	33	49,741	6,083	137,846	6,459	0,001	9,004	12,867	10,771
15	34	50,071	6,172	141,651	6,967	0,001	8,250	12,228	10,792
15	35	69,343	6,599	208,502	7,027	0,001	10,374	12,748	11,458
15	36	111,701	6,741	292,588	7,402	0,001	8,662	16,788	11,367
15	37	51,100	8,415	134,683	8,672	0,001	10,376	12,719	11,575
15	38	48,710	7,555	136,984	8,484	0,001	9,206	13,183	11,297
15	39	52,773	8,327	143,015	9,032	0,001	11,748	15,586	13,118
15	40	54,685	8,825	150,836	10,077	0,001	12,687	18,107	14,975
20	40	71,239	7,632	194,272	8,688	0,001	10,255	14,220	12,354
20	41	73,955	8,547	200,734	9,357	0,001	11,368	15,167	13,068
20	42	70,651	9,034	200,700	10,167	0,001	12,232	15,468	13,892
20	43	82,418	9,746	238,263	10,381	0,001	13,291	15,744	14,313
20	44	140,296	9,293	374,990	10,084	0,001	13,388	16,212	14,582
20	45	64,216	9,807	178,302	11,100	0,001	13,881	16,614	15,355
20	46	68,961	10,510	187,678	11,491	0,001	12,570	16,248	14,291
20	47	70,828	10,371	186,847	11,582	0,001	13,693	20,084	16,756
20	48	95,357	11,401	279,861	11,554	0,001	14,856	17,091	16,180
20	49	65,833	11,581	170,524	12,882	0,001	14,522	19,284	16,357
20	50	60,639	11,652	164,146	12,037	0,001	15,291	17,756	16,237
25	50	99,916	11,603	275,124	12,785	0,001	15,226	18,972	17,549
25	51	100,899	11,359	277,760	12,117	0,001	15,112	19,918	17,397
25	52	125,097	12,580	378,551	13,693	0,001	15,866	21,735	18,213
25	53	146,266	13,727	362,663	14,196	0,001	17,623	20,137	18,563
25	54	87,978	12,922	243,903	14,642	0,001	16,781	21,787	19,375
25	55	88,763	13,159	240,268	14,819	0,001	17,603	21,102	19,575
25	56	122,090	14,199	337,524	15,866	0,001	18,499	22,653	20,481
25	57	75,274	15,177	205,297	16,753	0,001	19,662	25,710	21,921
25	58	59,323	15,433	157,836	16,386	0,001	20,062	22,196	21,387
25	59	53,869	15,564	145,786	17,288	0,001	19,005	25,570	22,083
25	60	54,796	15,826	147,463	17,623	0,001	21,621	25,706	23,199
30	60	138,227	16,197	383,529	17,239	0,001	21,483	26,635	22,817
30	61	138,060	16,368	376,397	17,813	0,001	22,010	28,255	24,207
30	62	200,428	15,930	550,680	18,040	0,001	21,794	26,686	24,251
30	63	118,552	18,579	320,035	19,943	0,001	24,067	27,380	25,705
30	64	139,171	18,546	396,114	20,100	0,001	22,668	26,882	24,691
30	65	107,300	17,678	271,994	20,007	0,001	22,874	27,429	25,693
30	66	75,646	18,974	199,672	21,173	0,001	24,008	28,605	26,186
30	67	66,483	19,604	180,153	20,792	0,001	24,567	31,476	27,123
30	68	66,549	19,110	190,358	21,981	0,001	25,463	29,415	26,688
30	69	79,100	20,704	212,217	21,345	0,001	25,424	30,117	27,798
30	70	66,955	21,665	184,047	22,716	0,001	26,504	34,565	29,610

Considering only the performance of the allocation, that is its weight provided by weight function w, the results from the Table 5.15 are visualized in Figure 5.7.

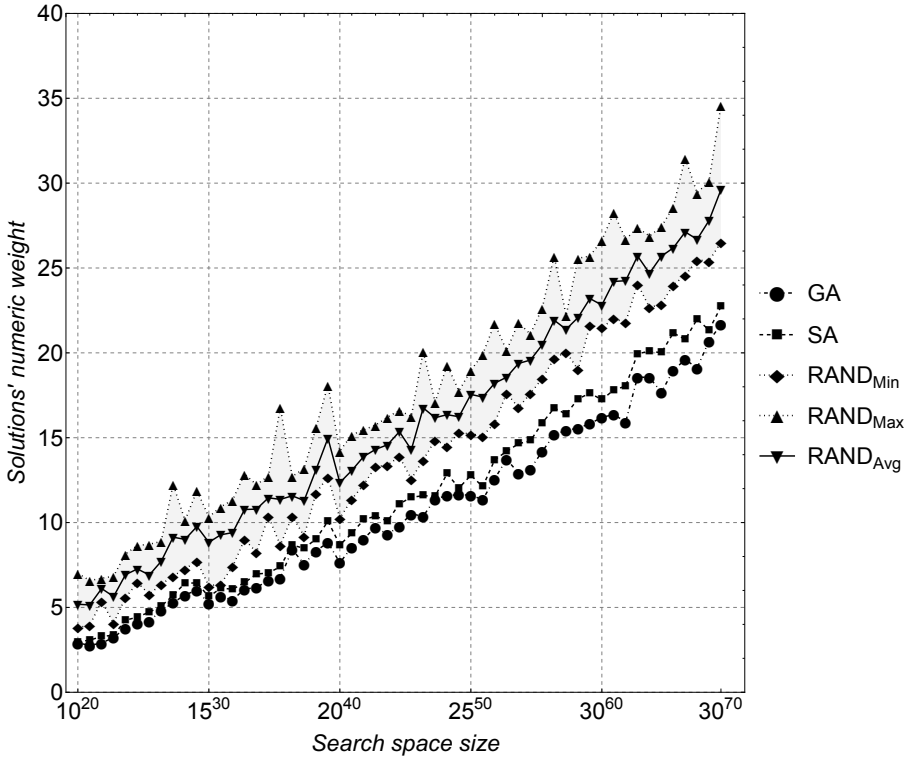

Figure 5.7: The performance of the allocations generated by GA, SA, and RAND as given by function w.

For each configuration, i.e. for each search space the GA provides allocations with the best performance, followed by allocations provided by the SA. The RAND however, provided the allocations with the worst performance, regardless whether it has the worst, best or average performing one among the 30 consecutively generated ones. Once again, the domain of the RAND allocations performance shaded in gray. Its best performing allocation got relatively closely to the performance of the ones generated by the GA and the SA around 15^{30}, however not in the single point did it perform better. With growing solution space its best solutions performed worse, for a relatively same amount. Comparing allocations obtained by the SA and the GA, it can be seen that the allocations obtained by the SA method performed well up until 15^{40}. Beyond that point its allocations mostly performed worse than the allocations obtained by the GA, however at several places it came fairly close.

Since the solutions obtained by the GA performed the best, an interesting comparison is illustrated in the Figure 5.8. It shows the difference in percentage between the best performing allocation provided by the GA in comparison to the performance of allocations provided by the SA and the RAND method. Although there are no surprises, this image offers a different perspective of the data. It reveals that in the most cases the SA allocations performed about 15% worse then the GA allocations. Sometimes

however it can get really close regardless of the search space. The most obvious thing to notice is that the RAND allocations have a fairly large difference between the best and worst performing one, and that with the increase of the search space this difference decreases. Furthermore, with increasing the search space, the difference between the best performing randomly generated solution and the solution obtained by the GA and SA gets narrower. The closest the best randomly generated allocation performance got to the allocations generated by the GA and the SA was for the search space of 15^{31}. At that point it was 14% worse than the GA solution, and only 4% worse than the SA solution. Nonetheless, even in best cases allocations given by the RAND method are on average 33% worse than the allocations obtained by the GA, and 22% worse than the allocations obtained by the SA, while on average it is on 55% worse than GA and 42% worse than SA.

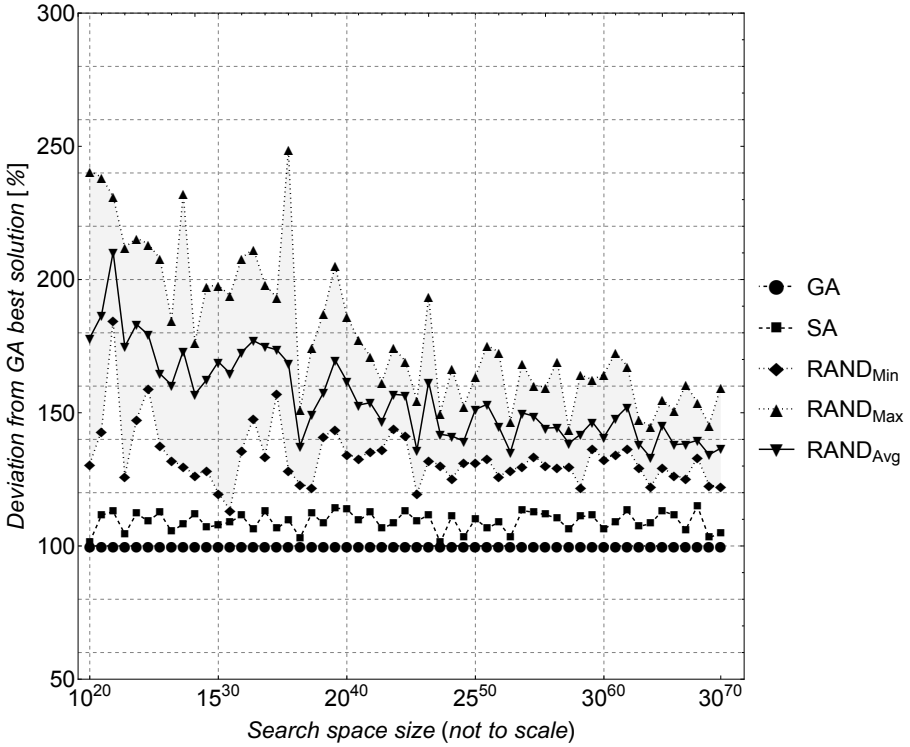

Figure 5.8: Difference in allocation performance as a offset from the GA solution in percent.

Given the large search spaces which are dealt with in this subsection, the processing time necessary to obtain the optimal allocation is an important consideration. Since the randomly generated allocations always take the same amount of time, they were left out of this comparison illustrated in Figure 5.9. As it can be seen, the processing time for obtaining the allocation is somewhat related to the search space size. Both the GA and SA methods processing time steadily increases with the raising search space. The GA method took less time to provide a solution than the SA did, almost in perfect proportion. At the beginning the time raises steadily, however toward the end the steady rise becomes unpredictable, thus resulting in outliers seen along the time curve. This is not attributed to search space size but to the measurement conditions. As for

the previous experiment, the data was processed on a non–dedicated server with no guarantee over its load. Hence, the repeated measurements have shown that there is no regularity in the appearing of these outliers, and as such, they are attributed to different processing load of the server and other OS processes. This was also verified on a dedicated platform which did not produce outliers but the results followed the same pattern. A final remark is given to the comparison of the processing time of the GA and the SA. For the maximal search space of 30^{70} the processing time was under 100 seconds for the GA approach and under 200 seconds for the SA approach, which is not nearly close close to 1 day and 14 hours the ES took to process a way smaller search space of 5^{12}.

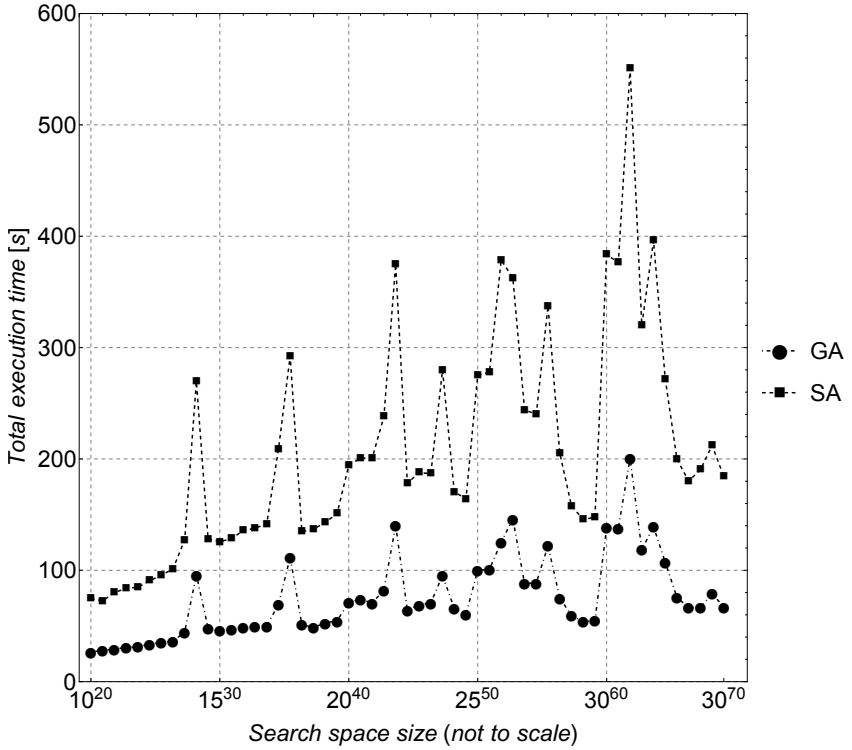

Figure 5.9: The time necessary to obtained the solution for large design spaces, compared between the GA and SA approach.

The conclusion of the experiments

The experiment results shown in the Table 5.15 reveal that not in a single configuration the randomly generated allocation outperformed an allocation obtained by the heuristic methods used in the I-IV allocation framework. Statistically, given a long enough time and much more tries, it should certainly produce an allocation which is the same or possibly better than the one produced by the I-IV framework, however for the 55 configurations in this experiment this did not occur. Therefore, it is concluded that the I-IV framework and its current implementation of optimizing the software allocation for large search spaces performs significantly better than the algorithm which

randomly generates software allocations and also it does so in a significant number of cases. Furthermore, it should be noted that the parameters of the GA and the SA were not reconfigured for each tested configuration, however tweaking them could probably produce in even better allocations in less time, but, as a proof–of–concept, both algorithms performed well.

With this, the experiments are complete and by the examination of the results by the tables 5.14 and 5.15 **the hypothesis H-2 is accepted**.

5.3 Summary

This chapter provides the verification of the I-IV allocation framework, a main contribution of this thesis in accordance to the research method shown in Figure 1.1.

The first hypothesis **H-1**, states that the framework for comparison of individual software component allocations correctly represents the system performance if the model for the heterogeneous computing platform is substantially valid. In order to verify this claim the following experiment involving the I-IV framework and its component allocation model α was proposed.
Given a set of real–world software components, computing units and different execution scenarios, the performance of an allocation as predicted by the model α should exactly match the performance of an allocation implemented and measured on a real–world platform. For that purpose, 6 different software allocations were selected, which represent 2 different execution scenarios. These allocations were evaluated by weighting function w from the model α and ranked by their performance. The procedure of measuring the inputs necessary for the model α is described in chapter 4.
Then, these allocations, i.e. software architectures were implemented on a real–world platform and their performance was measured. To be more precise, both power consumption and execution time were measured within the confidence interval of 95%. The resulting data was weighted by using a simplified AHP method presented in chapter 3, based on which the allocations were ranked by their performance. The resulting performance ranking obtained by measurements performed on a real–world platform perfectly aligned with allocation performance rankings obtained by the I-IV framework. Hence, the hypothesis **H-1** was accepted.

The second hypothesis, **H-2** states that in a case with large number of allocation choices, the I-IV allocation framework finds an allocation which has significantly better performance than randomly generated allocations. To verify this claim, the hypothesis was tested through two experiments.

The first one involved comparing the performance of the sub–optimal allocations generated by heuristic methods of I-IV framework with the performance of the optimal allocations obtained by performing an exhaustive search. The I-IV framework does not define the optimization method in the cases where exhaustive search is infeasible,

however its current implementation lets software architect choose between the genetic algorithm and the simulated annealing algorithm.

For the first experiment a number of heterogeneous platforms were generated and for each of them the best performing allocation was chosen, using the exhaustive search, the genetic algorithm, the simulated annealing algorithm and randomly generating allocations (RAND). The search space span between 2^8 and 5^{12}, where the base represents the number of computing units and the exponent represents the number of software components. Having that kind of search space, it is immediately clear that there is no known algorithm which could find the optimal solution in a polynomial time and that the sub–optimal solution would need to be satisfactory. For each generated heterogeneous platform, the optimal allocation given by the ES was compared against ones provided by the GA, the SA and the RAND in order to verify which method will provide the most accurate solution. Also, for each configuration, i.e. search space RAND was repeated 30 times and based on this 3 different allocations were selected, the average performing one, the best performing one and the worst performing one. The results have shown that the GA performed the best considering both the time it took to obtain an allocation and the performance of an allocation. On average, in all cases it took less than 10 seconds while the SA took between 10 and 21 seconds. In the worst case ES took more than 1 day and 14 hours.

From the standpoint of performance, in a worst case the allocation performance provided by GA was 7% worse than the optimal one, and for the SA this was 13%. The allocations obtained by RAND were at best closest to optimal solution with the 3% difference, but only for small search spaces. For larger search spaces this raised up to 22% and its worst allocation was 370% worse than the optimal allocation and on average it was 100% worse. Given these results the GA was accepted as an algorithm to provide allocations with satisfying performance, given both the time and precision.

For the second experiment the main interest was the performance of the GA in comparison to the RAND method for extremely large search spaces. The goal was to discover is it reasonable to use the GA (and SA) approach to find the optimal allocation of software components or would randomly generated allocations (RAND method) perform just as well. The search spaces to experiment with were ranging between 10^{20} and 30^{70} which are beyond the reach of the exhaustive search method and finding the optimal solution.

The results have shown that for such search spaces the GA provides the best performing allocations in the least amount of time, followed by the SA. Interestingly, the area between the worst and the best performing randomly generated solution was larger at the beginning and decreased towards the end of the defined search space. Regardless to which it never, not in a single case, outperformed the solution obtained by the GA nor the solution obtained by SA. At its best it was on average 33% worse than the allocation obtained by the GA and 22% worse than the allocation obtained by the SA. On average,

the allocation obtained by the RAND was 55% worse than the GA and 42% worse than the allocation obtained by SA.
Given the results from both the first and the second step of the experiment of testing the H-2 hypothesis, it was clearly shown that the allocations obtained by the I-IV framework are significantly better than the best randomly generated allocations in a significant number of cases. Statistically it should happen that, given a large enough search space, the RAND would produce at least one allocation which performs better than the one generated by I-IV framework. However in 55 different configurations tested here this did not occur, but even if it did it would be statistically insignificant. Therefore, the hypothesis H-2 was accepted.

CHAPTER 6

RELATED WORK

One of the main concerns of this thesis is software architecture optimization using a multi–criteria decision making process. This is a difficult task since it involves complex analysis and solving problems which very often classify as NP hard. Numerous researchers from different backgrounds such as mathematics, computer science, electrical engineering, etc. dealt and deal with this issue and their research resulted with many papers, which very often tackle the same issue but from different viewpoints. This chapter presents the work of other researchers closely related to this thesis.

6.1 Research classification

The research on the sheer complexity of the design decisions involved in making a software architecture for today's increasingly complex systems resulted in numerous papers from authors with various backgrounds. It is therefore not so easy to classify the related work, which was troubling for many researchers involved with allocation issues of high performance computing. To solve this mess, in 2014 Aleti et. al. performed a systematic literature review which dealt with classifying current software architecture optimization methods and approaches [3]. It was a laborious task which involved classifying 188 papers from different research communities, and as its authors report their investigation was guided by the following goals: a) classification framework and taxonomy for existing architecture optimization approaches, b) to provide a state of the art in this domain and c) to point out trends, gaps and research directions. Each of the papers is classified by the a) problem, b) solution and c) validation.

According to the classification framework by Aleti et. al.[3], the research presented in this thesis classifies as follows:

- **Problem category:** *performance optimization*
  - specific categorization: *performance, energy*
  - domain: *embedded systems*
  - design phase: *design time*
  - constraints: *general (cost, performance, mapping, etc.)*

- **Solution category:** *allocation*
  - degree of freedom: *allocation*
  - quality evaluation: *model based*
  - architecture representation: *optimization model, (custom architectural model, EMF)*
  - optimization strategy: *approximation (heuristic)*
  - constraint handling: *prohibit (for architectural constraints), penalty (for allocation performance)*
- **Verification category:** *experiment*

Comparing the research to papers analyzed in the aforementioned literature review, along with the latest publications (from ACM, IEEE Xplore, Google Scholar and Scopus) the most closely related work to this research deals with:

- *allocation, placement, mapping* – of processing tasks to different, very often, distributed computing resources.
- *heterogeneous system software architecture* – a very few research papers directly deal with heterogeneous systems and the ones which do, focus on distributed multi–processor systems. Only several publications deal with a context specific as CPU, GPU, FPGA, and the ones which do, concern with benchmarking performance of these computing units.
- *energy, power* – is a growing interest in software architecture community, mostly coming from the embedded systems and transiting to the new area of cyber–physical systems.

6.2 Software architecture optimization

The chapter 2 presents several different scientific fields which motivated this thesis. One can notice that the research closely related to this thesis is conducted by people from different backgrounds and that it is motivated by different, but generically similar problems. This section points out the closest related work associated with software architecture optimization. In particular, this section provides an overview of related research with the focus on input model types, optimization goals, model assumptions, degrees of freedom and results. The selected research papers are chosen based on their relevance to this research, experimentation and verification techniques, architectural input models, optimization algorithms and problem domain.

Koziolek and Reussner introduced a generic quality optimization framework for the design space exploration and optimizing any component based model for a number of quality properties and an arbitrary number of degrees of freedom [59]. This novel metamodel can support any component based metamodel, however they used OMG EMOF metamodeling language to represent input models. From such inputs, they derive the models automatically with a generic tool, independent of the component based

model applies an existing multi–objective optimization. With this metamodel called a Generic Degree of Freedom Metamodel, they can represent any changeable metamodel element. The values within the model represent the Degree of Freedom Instance which defines a design space in the form of the Cartesian product among different instances. The solution vector is obtained by evolutionary optimization.

Similarly, Malek et.al. developed an extensible framework for finding the most appropriate deployment architecture for a distributed software system with respect to multiple (conflicting) QoS[1] dimensions [68]. Their framework supports formal modeling of the allocation problem, and in their paper they evaluate it for precision and execution–time complexity. The authors focus on several QoS parameters, in particular, availability, latency, communication security, energy consumption and memory constraint. Their optimization strategy, i.e. the input model assumes that the software architect knows the constraints of the input parameters. The optimization was performed by a Mixed–Integer nonlinear Programming Algorithm (MINLP), a Greedy Algorithm and a Genetic Algorithm in order to maximize the overall objective function which is subject to a set of known constraints. This framework was verified by an experiment with 12 components, 5 deployment hosts and 8 different services. They report the Genetic Algorithm provided the best results, followed by the Greedy Algorithm and MINLP. In addition to everything said, an important characteristic of this framework is that it can be used in both run– and design– time.

Another, relatively generic framework is Sesame by Pimentel et. al. [88]. Sesame provides a high level modeling and simulation methods and tools for efficient system–level performance evaluation. Authors use a Y–chart[2] design based methodology for an early design space exploration. This framework includes the application models, mapping model, performance model and performance result. The constraints are represented as functions, and the goal of the architecture optimization is to identify a set of solutions which are superior in accordance to these functions. In their paper, authors focus on processing time, power consumption and total system cost, while the validation is made using Sesame's system level simulation environment which allows architectural exploration at different abstraction levels. A case study for Motion–JPEG encoder is made to clarify the model in detail.

The results of the previous work have shown that applying different optimization strategies has an impact to the time necessary to obtain the solution and to the result quality. A good comparison of the optimization strategies for the similar context is performed by Martens et. al. in their work presenting a hybrid approach for munlti–attribute QoS optimization [70]. In their paper authors proposed a combined use of analytical optimization techniques and evolutionary algorithms to efficiently identify a significant set of design alternatives, from which an architecture that best fits the differ-

[1] QoS – quality of service, the overall performance of a service provided by a system, often used in the context of telecommunication and networking

[2] It provides three perspectives on the system design, behavioral, structural and physical

ent quality objectives can be selected. The model for their work is based on the Palladio Component Model [8]. I uses input annotations for availability, cost and performance to describe the input model attributes. The optimization proceeds in three steps. The first step generates an initial candidate and a search problem formulation with consideration to degrees of freedom (component allocation, server configuration and component selection). In the second step, the solution is optimized using analytic techniques resulting in the Pareto–optimal solution candidates derived by Mixed–Integer Linear Programming (MILP). The third and final step uses the solution candidates obtained by the analytic techniques and performs evolutionary optimization. With the described model and procedure authors are able to predict system performance using SimuCom simulations and PerOpteryx. This hybrid approach proved to be superior to analytic optimization alone.

Predicting performance by applying the input parameters for the model has also been a subject to previous research. Martens et. al. also published a framework for automatically improving software models through quantitative prediction of quality properties, such as performance, reliability, and cost [71]. Their approach is most suited for composed based architecture (authors used Palladio) since such models encapsulate the functionality and can be independently used. As such, they are easy to manipulate with. The degrees of freedom in their paper were the processing speed, the number of servers, the components allocation and the component selection. The optimization step consisted of three steps. The first one was used to formulate the problem and derive a initial candidate. The second step derives the best solution candidates through evolutionary optimization while the third step presents the results. Candidate solutions are Pareto–optimal and they represent the prediction of the performance of a future system.

A work by Islam et. al. focuses more on distributed embedded systems, software component allocation, extra–functional requirements and both safety and non–safety critical operations. Unlike previously mentioned related papers, this one concerns specifically on mapping software components to hardware nodes with the goal to reduce error propagation [53]. Authors divide their problem in two stages, a) assigning software components to suitable hardware nodes and b) scheduling software execution. The assumption is that all the hardware nodes and software components can communicate with each other. To represent software components they used graph based approach, where the nodes represents software components which can be of two types, safety critical components and non–safety critical components. Additionally, the software components are subsequently decomposed in smaller units, i.e. jobs. The input models proposed by Islam et. al. provides the fault model for both hardware and software faults along with the constraints (binding constraints, dependability constraints, computing constraints, communication and timing constraints), which are represented by formal expressions. The suggested an 11 step algorithm which results with the mapping of software components onto hardware nodes uses the heuristic approach to obtain

the solution. To introduce data structures for support of job ordering and node availability, authors use a similar matrix based expressions as is used in this thesis. The presented algorithm guarantees the solution quality, reduces the search space and consistently provides the same mapping over many runs and thus being robust.

A very similar work related to allocation of tasks rather than components was made by El-Sayed et. al. Although the context is in borderline similarity to this research, the methodology applied resembles the one used in this thesis. El-Sayed et. al. presented a heuristic tool called Configuration Planner which determines the allocation and priority of processing tasks [33]. The input model of the software design can be in various forms but authors used Message Sequence Charts to display scenarios and UML Collaboration Diagram for describing object interactions. The model assumes a known resource requirements (represented by formal expressions), and the goal is to satisfy time constraints for each represented scenario. The allocation is found by using MULTIFIT-COM tool, which requires the knowledge of execution demand and communication overhead for each task in each scenario. Using a set of weight function, MULTIFIT-COM combines the execution cost and the communication cost to exploit the solution space and provides the task ranking. Further steps of the algorithm improve the task allocation until the algorithm is complete. The solution was verified using a statistical evaluation which involved generating a large sample of randomly generated systems. Their method succeeded in finding a feasible solution for more than 80% of samples tested, and depending on the input sample size they managed to raise the utilization of the CPU by 60–80% with 20–35 optimization iterations.

6.3 Software architecture physical footprint

The focus on the average power consumption presented in this thesis is mostly inspired by a currently growing trend in embedded systems, the cyber–physical theory. This perspective, characterizes software by its physical properties, which are apparent in its requirement and consumption of resources, e.g. time to execute, the power consumed, the generated heat, etc. [89, 120].

For software architects this presents additional considerations in the software design process. Component based software engineering facilitates techniques for expressing these system characteristics in the form of extra–functional properties [26], based on which software architects perform software design decisions.

A very important paper was made by Lee, in which he questions the fundamental approach to computing today [62]. Lee argues that the mismatch between abstractions for passage of time and concurrency of physical processes impede technical progress. Additionally, Lee suggests several research directions to make embedded systems easier to manage, control and predict. Lee calls for top–to–bottom rethinking of computation in order to identify computing abstractions that currently present obstacles for future

progress and which would span across multiple engineering disciplines, e.g. computer science, mechanical and electrical engineering. The main research suggestions focus on explicitly implementing timing in programming languages and embedding system predictability on all design levels (memory, concurrency, networking, pipelining, etc), etc.

Similar observations, about software leaving a physical footprint on the computing hardware are the subject of several chapters in the *Computers as Components: Principles of Embedded Computing System Design* book by Wolf [121]. He points out the power consumption as a particularly important design metric for battery–powered systems due to their limited lifetime. The power consumption can be optimized through several techniques without any intervention into hardware, but rather by applying certain software design principles. In his IEEE Software special issue publication, as Lee, Wolf calls for rethinking the design principles of embedded systems with consideration to software's physical footprint [120].

A research by Asano et al. reports that for image processing (2D filters), a GPU is by far the best platform, followed by a CPU, but only for filters up to a certain size, beyond which an FPGA surpasses both a CPU and a GPU [6]. However, the experiment results by Pauwels et al., show that for the real–time image processing an FPGA is without doubt the most suitable platform [86]. A comparison between a CPU and a GPU by Lee et al., showed that the performance gap between these platforms is not different in orders of magnitude as it is often considered [63].

For software architects this means that the current software models must support multiple criteria. There are several papers that take into consideration software profiling from the perspective of energy consumption [102], and a lot more from the perspective of execution time [62, 114], however additional research in this area is required for component based frameworks.

CHAPTER 7

CONCLUSION

This is the final chapter of this thesis which presents a brief overview of all previous chapters, emphasizes the research goals, answers the research questions and presents the promising directions for future research.

7.1 Model of the heterogeneous platform

The research presented in this thesis introduces the mathematical model for the formal description of a heterogeneous platform. This model, $\mathbb{M}_\alpha$ contains the information about the heterogeneous computing platform, the physical and the architecturally defined constraints, the allocation function and the set of parameters necessary to make a multi–criteria weighting function used for comparing the performance of different software component allocations.
As such, the component allocation model $\mathbb{M}_\alpha$ provides all the information necessary to attribute each feasible allocation of software components onto a heterogeneous computing platform (or homogeneous for that matter), with a number. This number enables the comparison of different allocations. An allocation with the lowest non–zero weight is also the best. The idea guiding the development of this model is to enable software architects to use it in the early system design phase and provide them an insight into the performance of a future system. Having this information, architectural decision making is largely simplified.
The procedure in which a software architect can obtain the best allocation consists of four steps:

I: define the heterogeneous platform $\mathbb{H}$, obtain the information about resource requirements and availability,

II: define the architectural constraints and (approximate / measure) the synergy effect trade–off array,

III: perform a pairwise resource comparison and calculate the resource importance trade–off vector,

IV: find an allocation $\alpha^{(a)}$ such that it is the lowest non–zero solution of the cost function w, i.e. $\min\left(w\left(\alpha^{(a)}\right)\right) > 0$.

The steps I-IV represent the framework for obtaining the (sub–) optimal allocation of software components onto a heterogeneous computing platform.

7.2 Measurement of software allocation performance

The data collected in this research suggests that with the respect to the average power consumption, the FPGA seems to be the best computing unit, followed by the CPU and the GPU. The GPU is the most power hungry component, however it also has the best data processing performance in respect with the average execution time. While the FPGA was very efficient in that regard as well, the end–to–end measurement of the average execution time increased its overall performance. With a SoC platform, having all three computing units on the same bus, results would be different, and therefore, claims stated in this research are platform specific, but the modeling and measurement techniques are general and reusable.

Regardless of the measurement results, determining the best computing platform proved to be very hard since the term *best* depends on the external circumstances, i.e. requirements. The main question at hand is to determine the best allocation of software components across different computing units to utilize the best aspects of all computing units, having in mind that different decision parameters can have different importance for different scenarios. Hence, it is this research suggests to use multi–criteria approach with strong emphasis on various execution scenarios. It was shown that different scenarios can have different performance and that considering them, a software architect can improve the architectural design of a system.

7.3 I-IV framework acceptability

Hypothesis H-1

The hypothesis **H-1**, states that the framework for comparison of individual software component allocations correctly represents the system performance if the model for the heterogeneous computing platform is substantially valid. In order to verify this the following experiment was proposed: given a set of real world software components, computing units and different execution scenarios, the performance predicted by the component allocation model $\mathbb{M}_\alpha$ should exactly match the performance of these components in real world obtained by measurements. For that purpose, six different software allocations were chosen which represent two different execution scenarios. These allocations were evaluated by the weight function w of the component allocation

model $\mathbb{M}_\alpha$ and ranked by their performance. Then, these allocations were implemented on a real–world platform and their performance was measured.

These six allocations were then ranked by the weight predicted by the component allocation model $\mathbb{M}_\alpha$, and by the data obtained by measurement. The ranks were perfectly aligned, and the hypothesis **H-1** was therefore accepted.

Hypothesis H-2

The hypothesis **H-2** claims that, given a large number of software components and computing units which results with a huge search space of possible allocations, the I-IV framework finds an allocation which has significantly better performance than a random allocation. To verify this statement, the hypothesis was tested in two steps.

The first step involved comparing the performance of sub–optimal allocations provided by the I-IV framework with performance of the optimal allocation. This is due to the fact that for large search space, one cannot use the exhaustive search method, and since a heuristic method needs to be used it is desirable to select the one which provides the allocations with minimal performance offsets from the optimal one. The I-IV framework does not define the optimization method, however its current implementation allows a software architect choose between the genetic algorithm (GA) and the simulated annealing algorithm (SA).
The solutions of these approaches were verified by generating random parameters for the $\mathbb{M}_\alpha$ model, i.e. random heterogeneous computing platforms, and for each of these, four different approaches were compared to obtain the best performing allocation, using: the exhaustive search (ES), the genetic algorithm (GA), the simulated annealing (SA) and three randomly generating allocations (the best, the worst and the average performing one) (RAND). The results have shown that the GA performed the best considering both the time it took to obtain the solution and its performance. From the standpoint of performance, in the worst case, the allocation provided by the GA performed 7% worse than the optimal one, and for the SA this was 13%. The allocations obtained by the RAND were at best closest to optimal allocation with the difference of 3%, but only for a very small search space. For larger search spaces this raised up to 22%, while its poorest allocation performance was 355% worse than the optimal allocation. On average it allocations generated by the RAND were performing 100% worse. Given these results GA was accepted as a descent algorithm to provide allocations with satisfying performance.

The second step involved verifying how do allocations obtained by the best performing heuristic method compare to randomly generated allocations with huge search spaces. More precisely, how would solutions generated by the GA compare to the best, the worst and the average performing randomly generated allocation. For this experiment, the search spaces were ranging between 10^{20} and 30^{70}, for which the exhaustive search is infeasible. The results have shown that in such search spaces the GA provides

the best performing allocations in the least amount of processing time.

Given the results from both the first and the second step of the experiment, i.e. the testing of the hypothesis **H2**, it is clearly shown that the allocations obtained by the I-IV framework are significantly better than randomly generated allocations in a significant number of case. Statistically it should happen that, given a large enough search space, the RAND approach would produce at least one allocation which performs better than the one generated by I-IV framework. However in 55 different configurations tested here this did not occur and therefore, the hypothesis **H-2** was accepted.

7.4 Research questions and research goals

The research conducted in this thesis was guided by three research goals and three research questions which were met and answered in previous chapters, although all of them were not explicitly emphasized. This subsection will emphasize the answers of the research questions and the research goals.

Research goals

The first research goal (**RG-1**) was to develop a model capable of describing a heterogeneous computing platform, including both software components and computing units of different types. This model could then be used to make performance predictions in the early design phase of a computing system.The component allocation model $\mathbb{M}_\alpha$ presented in this research consists of a heterogeneous computing platform $\mathbb{H}$ which includes both a set of software components $\mathcal{C}$ and a set of computing units $\mathcal{U}$, along with a set of resource constraints, a set of architectural constraints, trade-off vectors and a weighting function. Using all of these constructs, the performance of any particular allocation of software components across various computing units can be quantified.

Along with the component allocation model $\mathbb{M}_\alpha$, this thesis presents the I-IV framework which is used to generate the (sub–)optimal software allocation by minimizing the weight function w. This function is used to quantify and compare allocations in order to find the one which has the minimal, non–zero weight through a set of predefined steps, which was the second research goal (**RG-2**). The suggested I-IV allocation framework defines four steps for obtaining the optimal allocation of software components, these involve I) defining the heterogeneous platform and the resource constraints, II) defining the architectural constraints and the synergy effect trade-off, III), performing a pairwise resource comparison and IV) minimizing the weight function w.

However, the number of possible allocations, i.e. the solution space grows exponentially by adding new software components to the model. Therefore, the manual search is at best laborious, and in reality infeasible. Therefore, the third research goal **RG-3** suggested automating the I-IV framework with a software tool. Chapter 5 presents

SCALL – a software component allocator tool which uses heuristic algorithms to generate software allocations. The allocations obtained by SCALL are sub–optimal but acceptable, since the tool exploits a trade–off between the time necessary to obtain the allocation and the predicted performance of an allocation. SCALL is deployed as an Eclipse plug–in, with core algorithms developed in Python 3.5.

Research questions

The research question (**RQ-1**) asks how can one describe software components and heterogeneous computing platform using the same model? Furthermore how can this model represent communication separately from the processing parameters and how can this model incorporate architectural constraints? – This research question was a direct consequence of the first research goal. Therefore, the answer to this research question is the mathematical model presented in chapter 3. This model separates the processing performance parameters from the communication performance parameters in two cost functions. These cost functions can be written as one equation by using the AHP method for comparing phenomenons measured in different units. This single equation becomes a weighting function used to compare individual software allocations. Besides that, the architectural constraints defined by the software architect are incorporated in the model with matrices, array and constraint functions which describe the relation between components and the hosting capabilities of computing units.

The second research question (**RQ-2**) deals with measuring performance and communication intensity of particular software component allocated on a particular computing unit. The detailed answer is given in chapter 4. In short, the performance parameters which are measured are the average power consumption and the average execution time. Both of these are thoroughly measured for each software component allocated on multiple different computing units. The results of these measurements generated 183 files with almost half a million rows of raw data. All the values were measured and verified to satisfy the statistical confidence interval of 95%. The measured values represent the real–world performance of each software component allocated to each computing unit (with some exceptions). As such, these values were used to verify the correctness, i.e. the trustworthiness of the component allocation model $\mathbb{M}_\alpha$. Furthermore, since the communication was not the main concern of the measurements of this thesis, it does provide an answer on to how to quantify it. As for the performance, by using specialized software tools like VisualVM and considering different execution scenarios, one can measure the exact data delivery time, bandwidth usage and the communication intensity by counting the number of calls to each of the components.

The final research question (**RQ-3**) deals with optimizing the software architecture for which the full answer is provided in chapter 5. To obtain the optimal software allocation, four different methods were used and compared, 1) the exhaustive search, 2) the genetic algorithm, 3) the simulated annealing and 4) randomly generating software al-

locations. As expected, the exhaustive search provided the best performing allocations in comparison to other methods, but since this approach is not applicable for larger number of software components and computing units, other approaches were also considered. The genetic algorithm provided the best allocations in the means of the time necessary to obtain it and its performance. In addition it has been shown that in the significant number of occasions the proposed I-IV framework generates significantly better allocations than any randomly generated allocations.

7.5 Future work

Although this is the concluding chapter of this thesis, the research related to this subject is by no means complete, there are plenty of interesting research directions for future researchers, in particular:

- *communication minimization*, a subject of many researchers, specifically in the automotive and aircraft industry. Minimizing the communication between different components reduces vehicle wiring, consequently saving weight and reducing the communication latency. Since this research does not focus on the communication but merely considers it, it should be investigated in future.
- *research into heuristics*, it was shown that the allocations generated by heuristic approaches are trustworthy and perform better than randomly generated ones. However, during the verification the parameters of both approaches, (the genetic algorithm and the simulated annealing) were not tweaked to improve their performance. Therefore, it is possible that these approaches can be further improved to provide even better allocations in less time. Since for large number of software components and computing units the time necessary to obtain the result grows, it would be interesting to obtain allocations in the least possible amount of time to appeal for larger systems (e.g. cloud, docker).
- *dynamic systems*, the I-IV framework was designed for static analysis of a system int its early design phase, however some scenarios require fast, i.e. dynamic adaptations of the software architecture depending on the environmental changes. The future research should consider such scenarios, upgrade and verify the I-IV framework for dynamic software configuration.
- *improve FPGA component design*, due to the lack of available IP cores, this research focused more on software components for CPUs and GPUs. The rapid development of FPGA programming models and their recent integration into widely available computing environments will soon enough make FPGAs an essential part of any embedded computer. This has been seen with GPUs, so sooner or later it will happen with FPGAs. This puts forward the question of who gets to decide where does the software get to be executed? – While this research suggests the framework for architectural decision making, future research should consider more software

components which are truly independent of the computing environment and verify how good of an allocation can the I-IV framework produce.

- *improve SCALL and make it web oriented*, at this point SCALL is developed in Python, Eclipse and it is partially generated from the EMF model to conform to the development preferences of the modeling community. Since its main decision making algorithm (PyAllocator) is made entirely in Python and also completely decoupled from the visual representation of the model, in future it should be web oriented. A web application would provide better accessibility among software architect practitioners, where they would have a better chance to provide feedback. This would also make the tool completely platform independent.
- *consider different scenarios*, although I-IV framework was verified on a heterogeneous platform of a robot, it is in no way bounded purely to embedded systems. One of the major interests for its future development is to apply it to other scenarios, e.g. docker allocation, hardware–software codesign, integration with existing component models, etc.
- *lessons from cyber–physical theory*, this research, as did ones before it related to cyber–physical computing, shows that a software has its own physical footprint. In future investigation it would be interesting to find out to what end can a software allocation, architecture and architectural style influence the physical parameters of the embedded systems, mainly focusing on the energy consumption.

— EOF —

BIBLIOGRAPHY

[1] a.a. Khokhar, V. K. Prasanna, M. E. Shaaban, and C.-L. Wang. Heterogeneous computing: challenges and opportunities. *Computer*, 26(6):18–27, 1993.

[2] A. Adel and B. Abbdellah. Semantic mapping of ADLs into MDA platforms using a meta-ontology. *2010 International Conference on Machine and Web Intelligence*, pages 426–433, oct 2010.

[3] A. Aleti, B. Buhnova, L. Grunske, A. Koziolek, and I. Meedeniya. Software Architecture Optimization Methods: A Systematic Literature Review. *IEEE Transactions on Software Engineering*, 39(5):658–683, may 2013.

[4] AMD. Introduction to OpenCL. Technical report, 2012.

[5] Amd. AMD Accelerated Parallel Processing OpenCL Programming Guide. (November), 2013.

[6] S. Asano, T. Maruyama, and Y. Yamaguchi. Performance comparison of FPGA, GPU and CPU in image processing. *FPL 09: 19th International Conference on Field Programmable Logic and Applications*, pages 126–131, 2009.

[7] M. Baklouti, M. Ammar, P. Marquet, M. Abid, and J.-L. Dekeyser. A model-driven based framework for rapid parallel SoC FPGA prototyping. *2011 22nd IEEE International Symposium on Rapid System Prototyping*, pages 149–155, may 2011.

[8] S. Becker, H. Koziolek, and R. Reussner. The Palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82(1):3–22, 2009.

[9] B. Betkaoui, D. B. Thomas, and W. Luk. Comparing performance and energy efficiency of FPGAs and GPUs for high productivity computing. *2010 International Conference on Field-Programmable Technology*, pages 94–101, dec 2010.

[10] E. Blern and K. Sankaralingarn. Power Struggles: Revisiting the RISC vs. CISC Debate on Contemporary ARM and x86 Architectures. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pages 1–12, Shenzhen, 2013. IEEE Computer Society.

[11] G. Booch. The Big Questions. *IEEE Software*, 31(4):9–11, jul 2014.

[12] E. Borde, J. Carlson, J. Feljan, L. Lednicki, T. Lévêque, J. Maras, and A. Petrić. PRIDE – an Environment for Component-based Development of Distributed Real-time Embedded Systems. pages 351–354, 2011.

[13] U. Brinkschulte, A. Bechina, F. Picioroagii, and E. Schneider. Open system architecture for embedded control applications. In *Industrial Technology, 2003 IEEE*

International Conference on, pages 1247–1251, 2003.

[14] F. Brosch, H. Koziolek, I. C. Society, and B. Buhnova. Architecture-Based Reliability Prediction with the Palladio Component Model. 38(6):1319–1339, 2012.

[15] T. Bureš, J. Carlson, and I. Crnkovic. ProCom–the Progress Component Model Reference Manual. Technical report, 2008.

[16] O. Bushehrian and R. Baghnavi. Deployment optimization of software objects by design-level delay estimation. *The Journal of Supercomputing*, 65(3):1243–1263, jan 2013.

[17] D. Cancila, R. Passerone, T. Vardanega, and M. Panunzio. Toward Correctness in the Specification and Handling of Non-Functional Attributes of High-Integrity Real-Time Embedded Systems. *IEEE Transactions on Industrial Informatics*, 6(2):181–194, may 2010.

[18] J. Carlson, J. Feljan, J. Maki-Turja, and M. Sjodin. Deployment Modelling and Synthesis in a Component Model for Distributed Embedded Systems. *2010 36th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 74–82, sep 2010.

[19] T. Çelik and B. Tekinerdogan. S-IDE: A tool framework for optimizing deployment architecture of High Level Architecture based simulation systems. *Journal of Systems and Software*, 86(10):2520–2541, oct 2013.

[20] C. Chen, G. Novick, and K. Shimano. Risc vs cisc. http://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/about/index.html, 2000. Accessed: 2015-12-29.

[21] P. K. Chouhan, E. Caron, and F. Desprez. Automatic middleware deployment planning on heterogeneous platforms. *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–13, apr 2008.

[22] P. Clements and M. Shaw. " The Golden Age of Software Architecture" Revisited. *Software, IEEE*, pages 70–72, 2009.

[23] S. Collange, D. Defour, and A. Tisserand. Power consumption of GPUs from a software perspective. *Lecture Notes in Computer Science*, 5544 LNCS(PART 1):914–923, 2009.

[24] I. Corporation. Intel Multi-Core Processors: Making the Move to Quad-Core. Technical report, Intel Corporation, 2006.

[25] I. Crnkovic, M. Larsson, and O. Preiss. Concerning predictability in dependable component-based systems: Classification of quality attributes. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 3549 LNCS:257–278, 2005.

[26] I. Crnkovic, S. Sentilles, A. Vulgarakis, and M. R. V. Chaudron. A Classification Framework for Software Component Models. *Software Engineering, IEEE Transactions on*, 37(5):593–615, 2011.

[27] I. Crnkovic, J. Stafford, and C. Szyperski. Software components beyond pro-

gramming: From routines to services. *Software, IEEE*, pages 22–26, 2011.

[28] M. A. Cusumano, A. MacCormack, C. F. Kemerer, and W. Crandall. Critical decisions in software development: updating the state of the practice. *IEEE software*, pages 84–87, 2009.

[29] Y. Dai, M. Xie, K. Poh, and B. Yang. Optimal testing-resource allocation with genetic algorithm for modular software systems. *Journal of Systems and Software*, 66(1):47–55, apr 2003.

[30] M. de Jonge. Build-level components. *IEEE Transactions on Software Engineering*, 31(7):588–600, jul 2005.

[31] A. Dogru and M. Tanik. A process model for component-oriented software engineering. *Software, IEEE*, 2003.

[32] E. Dolif, M. Lombardi, and M. Ruggiero. Communication-aware stochastic allocation and scheduling framework for conditional task graphs in multi-processor systems-on-chip. *EMSOFT '07 Proceedings of the 7th ACM & IEEE international conference on Embedded software*, 13(4):315–345, 2007.

[33] H. El-Sayed, D. Cameron, and C. M. Woodside. Automation support for software performance engineering. *Proceedings of the Joint International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS/Performance)*, pages 301–311, 2001.

[34] European Commision. EN HORIZON 2020 WORK P ROGRAMME 2014 – 2015 Leadership in enabling and industrial technologies Information and Communication Technologies. 2015(July 2014), 2015.

[35] Y.-H. Fan, J.-O. Wu, and S.-F. Wang. Software synthesis of middleware for heterogeneous embedded systems. *2012 2nd International Conference on Consumer Electronics, Communications and Networks (CECNet)*, pages 2084–2087, apr 2012.

[36] F. Feinbube, P. Tröger, and A. Polze. Joint Forces: From Multithreaded Programming to GPU Computing. *Software, IEEE*, 2011.

[37] J. Feljan and J. Carlson. Task allocation optimization for multicore embedded systems. In *Software Engineering and Advanced Applications (SEAA), 2014 40th EUROMICRO Conference on*, pages 237–244. IEEE, 2014.

[38] J. Fern, L. Han, A. Nuñez, J. Carretero, and J. V. Hemert. Using Architectural Simulation Models to Aid the Design of Data Intensive Application. *2009 Third International Conference on Advanced Engineering Computing and Applications in Sciences*, pages 163–168, oct 2009.

[39] R. Feynman. *The Character of Physical Law*. The MIT Press, Cambridge Massachusetts, London England, 1965.

[40] F.-A. Fortunm, F.-M. De Rainville, M.-A. Gardner, M. Parizeau, and C. Gagn. DEAP : Evolutionary Algorithms Made Easy. *Journal of Machine Learning Research*, 13:2171–2175, 2012.

[41] J. Fowers, G. Brown, P. Cooke, and G. Stitt. A performance and energy comparison of FPGAs, GPUs, and multicores for sliding-window applications. *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays - FPGA '12*, page 47, 2012.

[42] E. Fykse. Performance Comparison of GPU , DSP and FPGA implementations of image processing and computer vision algorithms in embedded systems Egil Fykse. (June), 2013.

[43] B. Gaster, L. Howes, D. R. Kaeli, P. Mistry, and D. Schaa. *Heterogeneous Computing with OpenCL*. Elsevier Inc., first edition, 2012.

[44] C. Grozea, Z. Bankovic, and P. Laskov. FPGA vs . Multi-Core CPUs vs . GPUs : Hands-on Experience with a Sorting Application. pages 1–12.

[45] J. Grundy, W. Mugridge, and J. Hosking. Constructing component-based software engineering environments: issues and experiences. *Information and Software Technology*, 42(2):103–114, jan 2000.

[46] J. Hahn and P. Chou. Buffer optimization and dispatching scheme for embedded systems with behavioral transparency. *EMSOFT '07 Proceedings of the 7th, ACM & IEEE international conference on Embedded software*, 2007.

[47] M. C. Hause and F. Thom. An Integrated MDA Approach with SysML and UML. *13th IEEE International Conference on Engineering of Complex Computer Systems (iceccs 2008)*, pages 249–254, mar 2008.

[48] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Elsevier Inc., 5 edition, 2012.

[49] D. Hower. GPU Architectures A CPU Perspective Data Parallel Execution on GPUs. Technical report, AMD Research, 2013.

[50] C.-Y. Huang and J.-H. Lo. Optimal resource allocation for cost and reliability of modular software systems in the testing phase. *Journal of Systems and Software*, 79(5):653–664, may 2006.

[51] N. Ibrahim, M. Hassan, and Z. Balfagih. Agent-based MOM for interoperability cross-platform communication of SOA systems. In *Humanities, Science & Engineering Research (SHUSER), 2011 International Symposium on*, pages 40–45, Kuala Lumpur, 2011.

[52] IEEE Computer Society. *Guide to the Software Engineering Body of Knowledge Version 3.0 (SWEBOK Guide V3.0)*. IEEE Computer Society, 2013.

[53] S. Islam, R. Lindström, and N. Suri. Dependability driven integration of mixed criticality SW components. *Proceedings - Ninth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, ISORC 2006*, pages 485–495, 2006.

[54] T. B. Jablin, P. Prabhu, J. a. Jablin, N. P. Johnson, S. R. Beard, and D. I. August. Automatic CPU-GPU communication management and optimization. *ACM SIGPLAN Notices*, 47(6):142, aug 2012.

[55] D. H. Jones, A. Powell, C.-S. Bouganis, and P. Y. Cheung. GPU Versus FPGA for High Productivity Computing. *2010 International Conference on Field Programmable Logic and Applications*, pages 119–124, aug 2010.

[56] J. Keranen and T. Raty. Model-based testing of embedded systems in hardware in the loop environment. *IET Software*, 6(4):364, 2012.

[57] S. Kestur, J. D. Davis, and O. Williams. BLAS Comparison on FPGA,CPU and GPU. In *Proceedings - IEEE Annual Symposium on VLSI, ISVLSI 2010 (2010)*, pages 288–293, 2010.

[58] G. Kim, J. Park, and J. Baik. An effective approach to identifying optimal software reliability allocation with consideration of multiple constraints. *Proceedings - 2012 IEEE/ACIS 11th International Conference on Computer and Information Science, ICIS 2012*, pages 541–546, may 2012.

[59] A. Koziolek and R. Reussner. Towards a generic quality optimisation framework for component-based system models. *Proceedings of the 14th international ACM Sigsoft symposium on Component based software engineering*, pages 103–108, 2011.

[60] H. Koziolek. Performance evaluation of component-based software systems: A survey. *Performance Evaluation*, 2010.

[61] K.-K. Lau and Z. Wang. Software Component Models. *IEEE Transactions on Software Engineering*, 33(10):709–724, oct 2007.

[62] E. a. Lee. Cyber-Physical Systems - Are Computing Foundations Adequate ? In *Position Paper for NSF Workshop On Cyber-Physical Systems: Research Motivation, Techniques and Roadmap*, volume 1, pages 1–9. Citeseer, 2006.

[63] V. W. Lee, P. Hammarlund, R. Singhal, P. Dubey, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, and S. Chennupaty. Debunking the 100X GPU vs. CPU myth. *ACM SIGARCH Computer Architecture News*, 38(3):451, 2010.

[64] B. M. Levy and C. Promotions. The History of The ARM Architecture : From Inception to IPO. *Electronics*, pages 14–19, 1990.

[65] J. Li, N. T. Pilkington, F. Xie, and Q. Liu. Embedded Architecture Description Language. *2008 32nd Annual IEEE International Computer Software and Applications Conference*, pages 36–43, 2008.

[66] P. López Martínez, L. Barros, and J. M. Drake. Design of component-based real-time applications. *Journal of Systems and Software*, 86(2):449–467, feb 2013.

[67] D. Luebke and G. Humphreys. How GPUs work. *IEEE Computer*, 2007.

[68] S. Malek, N. Medvidovic, and M. Mikic-Rakic. An extensible framework for improving a distributed software system's deployment architecture. *IEEE Transactions on Software Engineering*, 38(1):73–100, 2012.

[69] F. Mallet, C. André, and J. DeAntoni. Executing AADL Models with UML/MARTE. *2009 14th IEEE International Conference on Engineering of Complex Computer*

Systems, pages 371–376, 2009.

[70] A. Martens, D. Ardagna, H. Koziolek, R. Mirandola, and R. Reussner. A hybrid approach for multi-attribute QoS optimisation in component based software systems. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6093 LNCS:84, 2010.

[71] A. Martens, H. Koziolek, S. Becker, and R. Reussner. Automatically improve software architecture models for performance, reliability, and cost using evolutionary algorithms. *Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering*, pages 105–116, 2010.

[72] M. Martin and R. Amir. CIS 501 (Fall 2005): Introduction To Computer Architecture - Course Lectures. Technical report, University of Pennsylvania, 2005.

[73] J. Martínez, P. Merino, A. Salmerón, and F. Malpartida. UML-Based Model-Driven Development for HSDPA Design. *IEEE Software*, 26(3):26–33, 2009.

[74] P. Marwedel. *Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems*. 2010.

[75] R. V. Massow, A. V. Hoorn, and W. Hasselbring. Performance simulation of runtime reconfigurable component-based software architectures. *Software Architecture*, (1015), 2011.

[76] B. Morel and P. Alexander. SPARTACAS: automating component reuse and adaptation. *Software Engineering, IEEE Transactions on*, 30(9):587–600, 2004.

[77] A. E. Mrabti, H. Sheibanyrad, F. Rousseau, F. Petrot, R. Lemaire, and J. Martin. Abstract Description of System Application and Hardware Architecture for Hardware/Software Code Generation. *2009 12th Euromicro Conference on Digital System Design, Architectures, Methods and Tools*, pages 567–574, aug 2009.

[78] J. C. Munson. *Software Engineering Measurement*. CRC Press, Inc., Boca Raton, FL, USA, 2002.

[79] Nato Rto Task Group. *Validation, Verification and Certification of Embedded Systems*, volume 323. 2005.

[80] J. Nickolls and D. Kirk. Graphics and computing gpus. computer organization and design,(patterson and hennessy), chapter a, pages a. 1–a. 77. 2009.

[81] Nvidia. Delivering Unparalleled Performance and. Technical Report November, 2006.

[82] OMG. The UML Profile for MARTE: Modeling and Analysis of Real-Time and Embedded Systems.

[83] I. Paik, T. Han, D. Oh, S. Ha, and D. Park. An affiliated search system for an electronic commerce and software component architecture. *Information and Software Technology*, 45(8):479–497, jun 2003.

[84] J. Palacios and J. Triska. A Comparison of Modern GPU and CPU Architectures: And the Common Convergence of Both. Technical report, 2011.

[85] M. Panunzio and T. Vardanega. A component-based process with separation of

concerns for the development of embedded real-time software systems. *Journal of Systems and Software*, 96:105–121, oct 2014.

[86] K. Pauwels, M. Tomasi, J. Díaz Alonso, E. Ros, and M. M. Van Hulle. A Comparison of FPGA and GPU for real-time phase-based optical flow, stereo, and local image features. *IEEE Transactions on Computers*, 61(7):999–1012, 2012.

[87] K. Philip, S. Masoudi, P. Eric, and H. Betty. Composing adaptive software. *Computer*, 37(7):56–64, 2004.

[88] A. D. Pimentel, C. Erbas, and S. Polstra. A systematic approach to exploring embedded system architectures at multiple abstraction levels. *IEEE Transactions on Computers*, 55(2):99–111, 2006.

[89] R. Poovendran. Cyber-physical systems: Close encounters between two parallel worlds. *Proceedings of the IEEE*, 98(8):1363–1366, 2010.

[90] T. Rauber and G. Runger. *Parallel programming*. Springer, 2010.

[91] K. Rege. Design patterns for component-oriented software development. *Proceedings 25th EUROMICRO Conference. Informatics: Theory and Practice for the New Millennium*, pages 220–228 vol.2, 1999.

[92] A. Repenning, A. Ioannidou, and M. Payton. Using components for rapid distributed software development. *Software, IEEE*, (April), 2001.

[93] F. Rincón, J. Barba, F. Moya, F. J. Villanueva, D. Villa, J. Dondo, and J. C. López. System-Level Middleware for Embedded Hardware and Software Communication.

[94] B. Ristau, T. Limberg, and G. Fettweis. A mapping framework for guided design space exploration of heterogeneous MP-SoCs. *Proceedings of the conference on Design, automation and test in Europe - DATE '08*, page 780, 2008.

[95] a. W. O. Rodrigues, F. Guyomarc'h, and J.-L. Dekeyser. An MDE Approach for Automatic Code Generation from UML/MARTE to OpenCL. *Computing in Science & Engineering*, 15(1):46–55, jan 2013.

[96] T. Saaty. What is the analytic hierarchy process? *Mathematical Models for Decision Support*, 48:109–121, 1988.

[97] T. Saaty. *Fundamentals of Decision Making and Prority Theory with the Analytic Hierarchy Process*. RWS Publications, 1994.

[98] T. L. Saaty. Decision-making with the AHP: Why is the principal eigenvector necessary. *European Journal of Operational Research*, 145(1):85–91, 2003.

[99] M. Sandrieser, S. Benkner, and S. Pllana. Explicit Platform Descriptions for Heterogeneous Many-Core Architectures. *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, 248481:1292–1299, may 2011.

[100] L. Santinelli, M. Chitnis, C. Nastasi, F. Checconi, G. Lipari, and P. Pagano. A component-based architecture for adaptive bandwidth allocation in wireless sensor networks. *Industrial Embedded Systems (SIES), 2010 International Symposium*

on, pages 1–10, 2010.

[101] B. Senouci, A. Kouadri.M, F. Rousseau, and F. Petrot. Multi-CPU/FPGA Platform Based Heterogeneous Multiprocessor Prototyping: New Challenges for Embedded Software Designers. *2008 The 19th IEEE/IFIP International Symposium on Rapid System Prototyping*, pages 41–47, jun 2008.

[102] C. Seo, S. Malek, and N. Medvidovic. An energy consumption framework for distributed java-based systems. *Proceedings of the twenty-second {IEEE/ACM} international conference on Automated software engineering*, pages 421–424, 2007.

[103] M. Shaw. Prospects for an engineering discipline of software. *Software, IEEE*, (November):15–24, 1990.

[104] M. Shaw. Writing Good Software Engineering Research Papers. pages 726–736, 2003.

[105] M. Shaw and P. Clements. The golden age of software architecture. *Software, IEEE*, (April):31–39, 2006.

[106] C. Shih, C.-T. Wu, C.-Y. Lin, P.-A. Hsiung, N.-L. Hsueh, C.-H. Chang, C.-S. Koong, and W. C. Chu. A Model-Driven Multicore Software Development Environment for Embedded System. *2009 33rd Annual IEEE International Computer Software and Applications Conference*, pages 261–268, 2009.

[107] A. Ssaed, W. Kadir, and S. Hashim. Metaheuristic Search Approach Based on In-house/Out-sourced Strategy to Solve Redundancy Allocation Problem in Component-Based Software Systems. *International Journal of Software Engineering and Its Applications*, 6(4):143–154, 2012.

[108] I. Švogor, I. Crnković, and N. Vrček. An extended model for multi-criteria software component allocation on a heterogeneous embedded platform. *Journal of Computing and Information Technology*, 21(4):211–222, 2013.

[109] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 2002.

[110] C. Szysperski. Component Technoligy - What, Where and How? In *International Conference on Software Engineering*, volume 6, 2003.

[111] J. Teich. Hardware/software codesign: The past, the present, and predicting the future. *Proceedings of the IEEE*, 100:1411–1430, 2012.

[112] D. B. Thomas, L. Howes, and W. Luk. A Comparison of CPUs , GPUs , FPGAs , and Massively Parallel Processor Arrays for Random Number Generation Imperial College London. pages 63–72, 2009.

[113] C. Trabelsi, S. Meftali, and J.-L. Dekeyser. Distributed control for reconfigurable FPGA systems: A high-level design approach. *7th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, pages 1–8, jul 2012.

[114] K. H. Tsoi and W. Luk. Power profiling and optimization for heterogeneous multi-core systems. *ACM SIGARCH Computer Architecture News*, 39(4):8, 2011.

[115] K. Urlich, G. Daniel, and D. Rolf. Toward Fully Automatic Synthesis of Embedded Software. *IEEE Embedded systems letters*, 2(3):53–57, 2010.

[116] V. Vyatkin. Software Engineering in Industrial Automation: State-of-the-Art Review. *Industrial Informatics, IEEE Transactions on*, 9(3):1234–1249, 2013.

[117] G. Wang, M. Di Natale, and A. Sangiovanni-Vincentelli. Optimal Synthesis of Communication Procedures in Real-Time Synchronous Reactive Models. *IEEE Transactions on Industrial Informatics*, 6(4):729–743, nov 2010.

[118] S. Wang and K. Shin. Early-stage performance modeling and its application for integrated embedded control software design. *ACM SIGSOFT Software Engineering Notes*, (Ic), 2004.

[119] J. Whittle, J. Hutchinson, and M. Rouncefield. The State of Practice in Model-Driven Engineering. *Software, IEEE*, pages 79–85, 2014.

[120] W. Wolf. Cyber-physical systems. *IEEE Computer*, 42(3):88–89, 2009.

[121] W. Wolf. *Computers as Components: Principles of Embedded Computing System Design*. Morgan Kaufmann, Elsiever, 2 edition, 2012.

[122] XILINX. Introduction to FPGA Design with Vivado High-Level Synthesis. Technical report, 2013.

[123] L. D. Xu. Enterprise systems: State-of-the-art and future trends. *IEEE Transactions on Industrial Informatics*, 7(4):630–640, 2011.

[124] X. Yixing and C. Yaowu. A Component-based Framework for Embedded Digital Instrumentation Software with Design Patterns. *Eighth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD 2007)*, (2003):166–170, jul 2007.

[125] A. M. Zaremski and J. M. Wing. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology*, 6(4):333–369, oct 1997.

[126] A. Ziani, B. Hamid, and S. Trujillo. Towards a Unified Meta-model for Resources-Constrained Embedded Systems. *2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 485–492, aug 2011.

Curriculum vitae

Ivan Švogor was born on 10th of October 1986 in Varaždin, Croatia, where in 2005 he finished high school. That year he also enrolled in University of Zagreb, Faculty of Organization and Informatics and in 2010 he majored *Information and Software Engineering* with cum laude honors. From 2010 to 2011 he worked as a software developer in the Center for Software Development at the aforementioned Faculty and in 2011 he became a research/teaching assistant, which he still is today. In late 2010 he enrolled with the doctoral study *Information Science* at University of Zagreb during which he spent one year as a *Ralf3* project associate at Mälardalen University, Sweden. His main research interests at the time are related to embedded software engineering (AUVs, ROVs, etc.) and cyber–physical systems.

List of publications

[1] I. Švogor, I. Crnković, N. Vrček. An Extended Model for Multi-Criteria Software Component Allocation on a Heterogeneous Embedded Platform. *Journal of Computing and Information Technology*, 21(4):211–222, 2013.

[2] I. Švogor. An initial performance review of software components for a heterogeneous computing platform. *ECSAW '15 Proceedings of the 2015 European Conference on Software Architecture Workshops*, ACM, 2015.

[3] I. Švogor; J. Carlson. SCALL: Software Component Allocator for Heterogeneous Embedded Systems. *ECSAW '15 Proceedings of the 2015 European Conference on Software Architecture Workshops*, ACM, 2015.

[4] I. Švogor, I. Crnković, N. Vrček. Multi–Criteria Software Component Allocation on a Heterogeneous Platform. *Proceedings of the ITI 2013 35th International Conference on Information Technology Interfaces*, IEEE Region 8, 2013.

[5] N. Vrček, I. Švogor, P. Vondra. Social Network Analysis and Software Evolution: A Perspective Method for Software Architecture Analysis? *Proceedings of the ITI 2013 35th International Conference on Information Technology Interfaces*, IEEE Region 8, 2013. t

[6] I. Švogor, T. Kišasondi. Two Factor Authentication using EEG Augmented Passwords. *Proceedings of the ITI 2012 34th International Conference on Information Technology Interfaces*, IEEE Region 8, 2012.

Source code of all developed programs (GitHub URL):
https://github.com/isvogor-foi/
Data collections:
https://dx.doi.org/10.6084/m9.figshare.c.2864134
05042016