Information and Software Technology 105 (2019) 30–42

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

An extensible framework for software configuration optimization on heterogeneous computing systems: Time and energy case study

Ivan Švogor[*,a,b], Ivica Crnković[c], Neven Vrček[b]

[a] Polytechnique Montréal, Montréal, QC, Canada
[b] Faculty of Organization and Informatics, University of Zagreb, Varaždin, Croatia
[c] Chalmers University of Technology, Gothenburg, Sweden

ARTICLE INFO

Keywords:
Cyber–physical systems
Software components
Power consumption
Execution time
Robot experiment
Heterogeneous computing
Component based software

ABSTRACT

Context: Application of component based software engineering methods to heterogeneous computing (HC) enables different software configurations to realize the same function with different non–functional properties (NFP). Finding the best software configuration with respect to multiple NFPs is a non–trivial task.

Objective: We propose a Software Component Allocation Framework (SCAF) with the goal to acquire a (sub–) optimal software configuration with respect to multiple NFPs, thus providing performance prediction of a software configuration in its early design phase. We focus on the software configuration optimization for the average energy consumption and average execution time.

Method: We validated SCAF through its instantiation on a real–world demonstrator and a simulation. Firstly, we verified the correctness of our model through comparing the performance prediction of six software configurations to the actual performance, obtained through extensive measurements with a confidence interval of 95%. Secondly, to demonstrate how SCAF scales up, we performed software configuration optimization on 55 generated use–cases (with solution spaces ranging from 10^{30} to 30^{70}) and benchmark the results against best performing random configurations.

Results: The performance of a configuration as predicted by our framework matched the configuration implemented and measured on a real–world platform. Furthermore, by applying the genetic algorithm and simulated annealing to the weight function given in SCAF, we obtain sub–optimal software configurations differing in performance at most 7% and 13% from the optimal configuration (respectfully).

Conclusion: SCAF is capable of correctly describing a HC platform and reliably predict the performance of software configuration in the early design phase. Automated in the form of an Eclipse plugin, SCAF allows software architects to model architectural constraints and preferences, acting as a multi–criterion software architecture decision support system. In addition to said, we also point out several interesting research directions, to further investigate and improve our approach.

1. Introduction

The recent increase of capacity and price decrease of heterogeneous computing platforms has enabled tremendous advancements in (real–time) data processing, primarily in artificial intelligence, robotics and cyber–physical computing. The appearance of software frameworks like OpenCL and programming language extensions which specifically target such systems promises seamless transition of code between a variety of platforms, e.g. CPUs, GPUs and FPGAs. This enables changing the execution environment of software components in design and run–time which makes the optimal utilization of such platforms questionable. In general, heterogeneous computing increases software complexity and makes optimizing system performance a challenging task, since it is not clear and straightforward for software architects how to choose the execution environment of software components. Furthermore, there can be numerous other functional and non–functional properties[1] on system design, as well as a plethora of constraints coming from either hardware design choices, software architecture itself, legacy components, etc. [1,2]. It is clear that in software architecture practice the number of decision variables can rapidly increase

* Corresponding author at: Department of Computer and Software Engineering, Montreal Polytechnic, Montreal, PQ H3T 1J4, Canada.
E-mail address: isvogor@foi.hr (I. Švogor).

[1] Non–functional properties describe the quality characteristics of a system and its influence as experienced by its users during its runtime. Typical examples include interoperability, performance, reusability, security, testabiltiy, throughput, etc.

https://doi.org/10.1016/j.infsof.2018.08.003
Received 29 August 2017; Received in revised form 5 July 2018; Accepted 9 August 2018
Available online 10 August 2018
0950-5849/

the design space of potential software architecture solutions, so the question is how to achieve the optimal software configuration with respect to multiple physically measurable properties? Furthermore, given the nature of heterogeneous computing environments, how can one incorporate energy consumption in software architecture design choices? For software architects this appoints new considerations which need to be taken into account in the (early) software design process.

Inspired by recent advocacy to rethink principles of embedded systems to incorporate considerations of software's physical footprint [3], in this paper we address software through its cyber–physical properties, which are apparent in its requirement and consumption of physical resources like the mean execution time, power consumption, dissipated heat, etc. [3,4]. To provide a systematic approach for such characterization of software we utilize the *component based software engineering* [5]. It is a suitable approach, which already facilitates techniques for expressing cyber–physical system characteristics in the form of non–functional properties [6–8]. In component based approach, software components are units of software composition with contractually defined interfaces which must be deployed independently (without concern of precedence and internal timing, but only dependencies). This allows software architects to exchange and configure software components much like one can exchange and configure hardware components on a computer, making software component configuration a degree–of–freedom in the system design process. We exploit the allocation of software components to various computing units as a system design degree–of–freedom to obtain an optimal or locally optimal software configuration with consideration to cyber–-physical software properties.

The rest of the paper is organized as follows; the following section presents the related work which served as inspiration for our research. Section 3 presents our research approach and problem statement. Section 4 presents our proposal for a framework which models the problem domain, while Section 5 presents our approach to software architecture optimization within this context. Section 6 describes the model instantiation, i.e. implementation. Sections 7 and 8 deal with the experiments which validate our approach and finally Section 9 concludes the paper.

2. Related work

As software optimization research is performed by researchers from various research communities, we present the related work in three main categories, (a) software architecture optimization, (b) software performance prediction and (c) low–level optimization of task mapping.

2.1. Software architecture optimization

In particular, this paper is inspired by the work of Koziolek and Reussner who introduced a generic quality optimization framework for the design space exploration, and optimization of any component based model for a number of quality properties and degrees of freedom [9]. Their novel approach used OMG EMOF meta–modeling language[2] to represent model inputs which were used for multi–objective software architecture optimization. The parameters in their model define the design space in the form of the Cartesian product among different instances of potential software architectures, while 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[3] dimensions [10]. The authors focus on availability, latency, communication security, energy consumption and memory constraint using the optimization with a Mixed–Integer nonlinear Programming Algorithm (MINLP), a Greedy Algorithm and a Genetic Algorithm. They report the Genetic Algorithm provided the best results, followed by the Greedy Algorithm and MINLP.

Another generic framework is Sesame by Pimentel et. al. [11]. It 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.

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. Their work presents a hybrid approach for mult–attribute QoS optimization [12] which is 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 different quality objectives can be selected. The model for their work is based on the Palladio Component Model [13]. It uses input annotations for availability, cost and performance to describe the input model attributes and 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. This hybrid approach proved to be superior to analytic optimization alone.

2.2. Performance prediction

Predicting performance by applying the input parameters for the model has also been a subject to previous research. Martens et. al. published a framework for automatically improving software models through quantitative prediction of quality properties, such as performance, reliability, and cost [14]. Their approach is most suited for component 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 time, 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 an 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, non–functional properties 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 [15]. 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 suggested eleven step algorithm which results with the mapping of software components onto hardware nodes uses a heuristic approach to obtain the solution.

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

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

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 paper. El-Sayed et. al. presented a heuristic tool called Configuration Planner which determines the allocation and priority of processing tasks [16]. 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 functions, 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.

2.3. Low–level optimization of task mapping

An important perspective on the component allocation issue comes from the perspective of the computer engineering community, which usually considers timing, scheduling and task mapping to specific processors. While the software component based approach operates on a different level of abstraction, the computer engineering community uses similar methods and approaches for solving the allocation, i.e. mapping problems.

In a literature review by Singh et. al. it is evident that challenges in task mapping are similar to software component allocation, however with different concerns. Mainly, meeting timing deadlines and optimizing for latency, delay, throughput, exploration time, etc.[17]. However, the optimization techniques that are being used are the same, e.g. simulated annealing, genetic algorithm, integer linear programming, etc.

Murali et al. [18] present a methodology that handles mapping of multiple use–cases while satisfying their performance constraints. Their methodology shows a power savings of 54%. Rhee et al. [19] propose an ILP based approach that optimally maps cores onto mesh architecture in order to minimize energy consumption. The approach achieves 81% energy savings for random benchmarks. In addition to processing, communication is an aspect often overlooked in current analysis. Hu et al. [20] propose a mapping methodology that reduces power consumption by decreasing the energy consumption in communication while guaranteeing the required performance. Their methodology provides an energy savings of 51%. Marcon et al. [21] extend the work in [37] and propose a technique called Communication Dependence and Computation Model (CDCM). This extension enabled reduction of execution time by 98% while achieving a significant amount of energy savings. Ascia et al. [22] present a GA based approach that explore Pareto mappings efficiently and accurately while optimizing for performance and energy consumption.

From the hardware–software integration standpoint, Paolucci et.al. developed a software–hardware architecture platform for embedded systems (SHAPES) which uses tiled architecture consisting of processing tiles connected by short wires [23]. Such tiles consist of RISC, VLIW, DSP, DNP, on–tile memories and peripherals. The goal of this framework from the software perspective is to provide an efficient programming environment for tiled architecture and seamless interfacing with re–configurable logic and signal acquisition and generation systems. This was used by Theile et.al. for distributed operation layer (DOL) which enables (semi–) automated mapping of applications onto the multiprocessor platform following SHAPES architecture. It optimizes for computation and communication time on a very low level. Furthermore it integrates an analytic performance analysis strategy into DOL to alleviate the modeling and analysis of system [24].

In 2014 Aleti et. al. performed a systematic literature review which dealt with analysis of current software architecture optimization methods and approaches [25]. It was a laborious task which involved classifying 188 papers from different research communities by its a) problem category, b) solution category and c) validation. Using the proposed classification we classify this paper as follows: *problem category* of this paper is software performance optimization in design time with constraints of energy and timing, the *solution category* is model based allocation with heuristic optimization strategy and the *verification* is an experiment.

In the next section, we intorduce our research approach to the previously mentioned issues, which leads to advances beyond related work.

3. Research approach

In this paper we present the Software Component Allocation Framework (SCAF), a theoretical model and its instantiation which addresses the aforementioned challenges. Our framework is capable of delivering an insight into the system performance in its early design phase, even before the components exist, and also of optimizing the software architecture for a heterogeneous computing platform in respect to multiple non–functional properties (NFP). Having said that, we emphasise that SCAF is a static approach and it consists of a) a formal description of component–based software system and a heterogeneous platform and b) formal identification of a cost function representing overall and generalized cost in respect to utilization of non–functional properties of software components. While the model is relatively simple, its implementation can be very complex, due to difficulties in specification of NFPs and obtaining their values. Therefore, SCAF is intended to be used during the early design phase of systems where knowing the necessary NFP values is of utmost importance, i.e. for highly critical software systems, where high levels of safety and integrity are required.

SCAF is a model and it enables different implementations depending on the non–functional properties of interest. Here we present a) the model itself, and b) the model instantiation used for validation.

3.1. The framework model

The framework model captures the properties of software components, the characteristics of the computing units (e.g. a CPU, GPU, FPGA, etc.) on a heterogeneous computing system, and defines the cost function for component executions. Also, it provides an optimization method of the software configuration to improve the overall system performance by allocating software components across various computing units. Since the allocation task is a form of *quadratic assignment problem*, which is a well known NP–hard problem [26], SCAF uses heuristic optimization methods.

3.2. The model instantiation

The model instantiation represents an instantiation of the framework for a set of selected non–functional properties, along with the methods of collecting their values. The selection of the NFPs for this paper was based on cyber–physical domain [3,4], which as previously stated characterizes software through its physical footprint [27]. The ones in which we focus in this research are:

a) *the mean execution time* – in particular, our research deals with platform–dependent mean execution time as the most important non–functional property related to the performance of a system.
b) *the average power consumption* – it directly affects major system design decisions; e.g. power supply size, cooling system, voltage regulators, etc. These influence the overall system size, the processing

power it can handle, durability, reliability, etc. [28].

For the validation of the framework, we performed the necessary steps to collect the input data necessary for the proposed theoretical model and produced a thorough framework demonstrator a tracked robot rover equipped with a CPU, a GPU and an FPGA.

4. The framework model

For a heterogeneous computing platform, the function of software is realized by allocating, i.e. deploying software components across the variety of available computing units. Thus, producing different configurations while preserving the functionality and changing the non–-functional properties. By exploiting this feature software allocation can be used as a design degree–of–freedom in order to increase the overall performance of a heterogeneous computing system with regard to multiple NFPs. This section presents the model of these features, which is utilized to provide the framework for optimizing software configurations in heterogeneous computing systems.

We define a heterogeneous computing system as a set of n software components $(c_i \in C,\ i = 0,\ ...,n)$ and a set of m computing units $(u_i \in \mathcal{U},\ i = 0,\ ...,m)$, where a configuration is the mapping of the set of components C to a (sub–) set of computing units $\mathcal{U}$. The configurations are permutations generated by the *component allocation function* α: $C \rightarrow \mathcal{U}$, where $\alpha^{(a)} = (p_1,...,p_n)$ refers to a particular ath configuration vector. The value of its element represents a computing unit, while its index corresponds to the software component. The total number of possible configurations is m^n, meaning that the configuration space grows rapidly and that finding the optimal configuration is not feasible in larger scale.

To quantify and evaluate different software configurations, we propose a cost function (w), which considers the computing resources[4] it requires (res) along with the physical and architectural constraints (ctr):

$$w(\alpha^{(a)}) = \mathrm{res}(\alpha^{(a)})\cdot \mathrm{ctr}(\alpha^{(a)}), \tag{1}$$

4.1. The computing resource cost function – $res(\alpha^{(a)})$

The computing resource cost function is a normalized sum of all resources used by all the components deployed to particular computing units. This sum changes depending on the deployment since the components behave differently on different computing units and due to the fact that not all resources are always equally important. To produce the sum of all resources, consider a heterogeneous computing system with n software components, m computing units, and l different computing resources. A resource consumption matrix $\mathcal{T} = [t_{ijk}]_{(n\times m\times l)}$ defines the resource necessity; its element t_{ijk}, represents the requirement on kth resource by the ith software component allocated to the $j$the computing unit.

However, computing units can host multiple components, and the measurements have shown that due to the resource sharing, the final resource requirement of a configuration does not always equal to the sum of individual requirements. In fact, in some cases having more components on the same unit requires more resources than the sum of their individual demands, while in others they require less. To the former we refer to as a *negative synergy effect* and to the latter we refer to as a *positive synergy effect*. To deal with this, in the model we introduced a synergy effect approximation matrix $\mathcal{S} = [s_{yjk}]_{(n\times m\times l)}$ (which is obtained by measurement or approximated by the experienced software architect). Its element s_{yjk}, represents a factor which compensates for the synergy effect for the kth resource, if the jth computing unit hosts y software components. If $s_{yjk} < 1$, the synergy effect is positive, or if its otherwise it is a negative synergy effect, i.e. $s_{yjk} > 1$, by default $s_{yjk} = 1$. The corresponding look-up function[5] which evaluates this effects is given as $\eta(\alpha^{(a)}, j) = \sum_{i=1}^{n} [p_j = p_i]$.

It is also important to normalize the previous sum and to account for the importance of different NFPs. For instance, a system designer could give more importance to energy consumption than to memory consumption or the execution time which are also measured in different units. So a cost function needs to provide a way to trade–off the influence of individual non–functional properties for the final allocation decision and to handle different measurement units. To deal with this, we use the *analytic hierarchy process* (AHP), an empirically proven and widely known technique for structuring and analyzing complex decisions [29]. In this particular case, AHP resolves the issue of orders of magnitude and the measurement units which are different for different NFPs with normalization. It also enables a non–bias procedure for obtaining a trade–off vector F used to define the importance of particular NFPs.

To apply AHP to making architectural decisions about allocating software components on a heterogeneous computing system, one level of hierarchy is sufficient. The importance of each resource is weighted by a software architect using a trade–off vector F. The values it contains are calculated in tree steps[6]:

1) Software architect performs a pairwise resource comparison in the form of a comparison matrix M_c which contains all k resources. The elements of the matrix provide favors between resources in the scale[7] from 1 to 9.
2) Calculate eigenvalues and select the biggest one, which is in AHP also known as the principal eigenvalue. Its corresponding eigenvector, the principal eigenvector, is a trade–off vector F. This vector contains weights of different criteria, or in this case to the weights of different NFPs (i.e. computing resource importance). With the introduction of F to Eq. (1), the cost function w becomes multi–-criterion.
3) Assess the consistency of pairwise comparison, since the pairwise comparison is subjected to human judgment it is prone to inconsistency. AHP deals with this using a *consistency ratio* which checks consistency of prioritization in the comparison matrix M_c.

Finally, given the previous definitions, to evaluate resource requirement of any particular software configuration we use the function $res(\alpha^{(a)})$, given as:

$$\mathrm{res}(\alpha^{(a)}) = \sum_{k=1}^{l} f_k \sum_{i=1}^{n} \left(t_{ip_ik}\cdot s_{\eta(\alpha,i),i,k}\right), \tag{2}$$

where t is the element of the resource consumption matrix $\mathcal{T}$, l is the number of different resources presented in matrix $\mathcal{T}$, n is the number of components, a is the a–th allocation vector, p_i is the i–th element of $\alpha^{(a)}$, η is the function which provides the synergy effect factor, and f_k is the trade-off vector for each resource.

4.2. The constraint function – $ctr(\alpha^{(a)})$

Due to limited resources of a heterogeneous computing system, or a design decision by a system designer, some configurations generated by the function α are not feasible for implementation. To deal with this, we introduce the constraint function $ctr(\alpha^{(a)})$ composed of four sub-functions with a codomain {0,1}: 1) ρ, the resource constraint function 2) δ,

[4] Refers to a physical or virtual resource with limited availability, e.g. computational time, memory, storage, etc.

[5] Where [ ] represent the Iverson's bracket notation, defined as $[P] = 1$ if P is true.

[6] For further and deeper understating of AHP we point the reader to [29].

[7] Standard AHP scale is interpreted as follows: 1 – equal importance, 3 – slightly favoring, 5 – strong favors, 7 – very strong favoring, 9 – extreme favors.

the hosting capability constraint function, 3) υ, the mandatory joint allocation constraint function, and 4) χ, the forbidden joint allocation constraint function.

ρ refers to the physical resource constraints and it is given implicitly through the previously defined elements of a heterogeneous computing platform's model. The remaining three functions δ, υ and χ are defined by a software architect and are used to account for various factors, e.g. involvement of legacy systems, immutable parts of the system, reuse of existing components, etc.

4.2.1. Resource constraint function ρ

It is used to discard configurations which require more resources than a platform can provide. To represent the resource availability we use the available resource matrix $\mathcal{R} = [r_{jk}]_{(m\times l)}$, where its element r_{jk} represents the availability of the kth resource on the jth computing unit. Hence, the function ρ results with 1 if the amount of processing resources is sufficient, i.e. if $\sum_{i=1}^{n}\sum_{k=1}^{l}(t_{ip_ik}) \leq \sum_{j=1}^{l} r_{p_ij}$, and with 0 if otherwise.

4.2.2. Hosting capability constraint function δ

It is used to specify whether the ith software component can be hosted on the jth computing unit. This specification is stored in the matrix $\mathcal{D} = [d_{ij}]_{(n\times m)}$. The value of its element d_{ij} can be either 1 or 0, the former if the ith software component cannot be allocated to jth and the latter if otherwise. For this constraint to be verified, each pair of elements of a configuration vector $\alpha^{(a)} = (p_1,...,p_n)$ is evaluated against the matrix $\mathcal{D}$ using the constraint function δ that results with 1 if $\sum_{i=1}^{n}(d_{ip_i}) = 0$ (meaning that all components are allocated to the allowed computing units) and with 0 if otherwise.

4.2.3. Mandatory joint allocation constraint function υ

It is used to specify which components should be allocated together, regardless of the computing unit. This is specified through the matrix $\mathcal{Y} = [y_{ij}]_{(n\times n)}$, where y_{ij} represents a statement whether the ith and jth software component should be allocated on the same computing unit. The value y_{ij} of the matrix $\mathcal{Y}$ is 1 if ith and jth component must be allocated on the same computing unit, and 0 if otherwise. The function υ which evaluates this constraint results with 1 if $\sum_{i<j} \neg\,[\,\neg\, y_{ij} \wedge [p_i = p_j]] = 0$, or with 0 if otherwise.

4.2.4. Forbidden joint allocation constraint function χ

It is used to specify the opposite of the previous constraint function, i.e. to specify which components must never be allocated on the same computing unit. For that purpose the matrix $\mathcal{X} = [x_{ij}]_{n\times n}$ was introduced. Its element x_{ij} represents a statement whether the ith and jth software component must not be allocated on the same computing unit. An element of the matrix $\mathcal{X}$ is 1 if ith and jth component must be allocated on separate computing units or 0 if otherwise. A function χ which evaluates forbidden joint allocations results with 1 if $\sum_{i<j}[x_{ij} \wedge [p_i = p_j]] = 0$, or with 0 if otherwise [8]. Finally, the resulting constraint function is defined as:

$$\mathrm{ctr}(\alpha^{(a)}) = \rho(\alpha^{(a)})\cdot\delta(\alpha^{(a)})\cdot\upsilon(\alpha^{(a)})\cdot\chi(\alpha^{(a)}) \tag{3}$$

ctr dismisses any configuration which does not satisfy the defined constraints.

5. Component allocation framework

Along with the elements of the modeling framework, the previous section also defined the cost function $w(\cdot)$ which provides a way of quantifying and evaluating the qualitative properties of each particular software configuration, with respect to multiple weighted criteria. With these, this brief section defines a procedure for finding the best performing software configuration on a given heterogeneous computing platform, considering the available resources, importance of different NFPs and additionally given constraints.

The *Software Component Allocation Framework* defines the following steps:

1. Define a selection of non–functional properties as requirements according to which the software architecture should be optimized.
2. By approximation, measurement or specification of component's non–functional properties and platform resources, obtain the information about resource requirement and availability for a specific heterogeneous computing system: populate the Resource consumption matrix $\mathcal{T}$ and the Resource availability matrix $\mathcal{R}$.
3. Specify the architectural constraints through the Hosting capability matrix $\mathcal{D}$, the Mandatory joint allocation matrix $\mathcal{Y}$ and the Forbidden joint allocation matrix $\mathcal{X}$, and the Synergy effect trade–off matrix $\mathcal{S}$.
4. Perform a pairwise resource comparison using the AHP method to obtain the NFP trade–off vector F.
5. Find a configuration $\alpha^{(a)}$ with a lowest non–zero value of the cost function; $\min(w(\alpha^{(a)})) > 0$. For large design spaces, i.e. those where the exhaustive search methods are infeasible, find the sub–optimal configuration using a heuristic method.

In order to help software architects in dealing with software allocation issues, we implemented SCAF as a software tool. This is further discussed in the later Section 8.2, while in the following section we examplify SCAF usage through a real–world demonstrator.

6. The model instantiation

This section presents the instantiation of the SCAF as defined in previous section. It includes (a) the selection of non–functional properties to be considered by the architecture optimization process, (b) measuring the values of non–functional properties, i.e. performing software component (time and energy) profiling and (c) setting the weight factors related to architectural decisions. In its original form, all the collected datasets are available on the Figshare open science data database[9].

6.1. The heterogeneous computing platform

For the data collection purpose and framework validation we used a custom built robot rover with an on board heterogeneous computing system, consisting of two integral parts; the hardware and the software.

6.1.1. Hardware

The computing hardware is built into a custom tracked robot assembled from a CNC milled steel chassis and 3D printed industrial grade plastics designed to withhold the maximum carrying weight of 20 kg. The electronics system is divided in two layers, (a) *the lower layer* which consists of a powertrain system and power supply system, and (b) *the upper layer* which is the heterogeneous computing system. This paper will only address the upper layer since it contains the heterogeneous computing system and therefore is the most relevant one for this research. It consists of:

– CPU: *Intel i3–3240 with 3.4GHz,*
– GPU: *low profile Sapphire Radeon HD7750,*
– FPGA: *Xilinx Spartan 6* (LogiPi board[10]).

[8] The relationship between matrices $\mathcal{Y}$ and $\mathcal{X}$ according to which $y_{ij} \in \mathcal{Y}$ and $x_{ij} \in \mathcal{S}$ follows that $y_{ij} = \neg\, x_{ij}$ only if $y_{ij} = 1$, for $i, j = 1, ...,n$.

[9] https://figshare.com/collections/I_IV_Allocation_Framework/2864134/1, *accessed in August, 2017.*

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

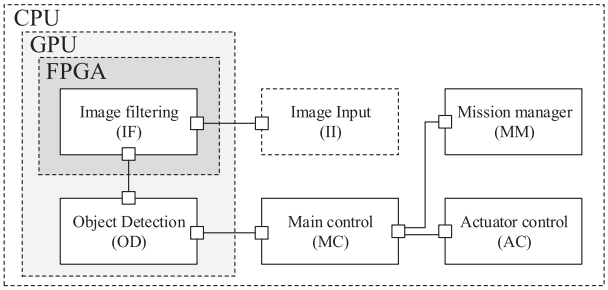

Fig. 1. Simplified software architecture layout, Image filtering component can be allocated to all computing units.

These were selected with the consideration to minimize the power consumption with the least sacrifice to performance, which is an ongoing struggle of the real world robots and their applications (e.g. autonomous or remotely operated areal vehicles, distributed sensor arrays, exploration robot swarms, etc.).

6.1.2. Software

Fig. 1 outlines the simplified layout of the software architecture in the form of a *Venn diagram* showing the component hosting capabilities of the computing units.

Obviously, the components of the most interest for this research are the *Image filtering* (IF) and the *Object detection* (OD) component, since their sub–components can be allocated to multiple different types of computing units of the specified HCP.

The Image filtering component consists of sub–components which correspond to the following image processing operations: Sobel (S), Gauss (G), Erode (E), Dilate (D) and Histogram equalization (H). Each of these components can handle five different input parameters, i.e. image sizes which affect the necessary computing power for efficient processing. These are QVGA (320 × 240), VGA (640 × 480), SXGA (1280 × 1024), FHD (1920 × 1080) and 8KHD (8192 × 8192). Due to the limitations of available FPGA intellectual property cores, it was limited to QVGA images. Each sub–component can be independently allocated and essentially represents a single independent building block of the software.

The architectural granularity of these components allows for 25 different image filtering forms, which can be chosen by a system designer. In image processing, these are often sequentially organized and referred to as *image processing pipeline*. Similarly, the Object detection component handles two different image inputs; VGA and FHD which further extends the design choices and is discussed further later on.

Both Image filtering and the Object detection component we implemented for a CPU, a GPU and an FPGA, and to overcome the challenge of multiple paradigms, we organized their internal architecture in multiple layers, spanning from the lowest ones implemented in C/C++ and Verilog, to the up most layers implemented in Java. The final software architecture can be composed of any number of these components organized sequentionally. Furthermore, for the CPU and GPU impelmentation the C++ code was the same thaks to the OpenCL. It allowed the code to be optimized (e.g. auto–vectorization for CPU's Advanced Vector Extensions) during compilation depending on build flags which determined the targeted platform.

6.2. Selection of non–functional properties and the measurement process

From this point forward we will address two non– functional properties in particular, the mean execution time, and the average power consumption. By no means are these two non–functional properties the only, or the most important ones which can be presented by SCAF. However, since these reflect the performance of a system and influence major design decisions, their consideration for this paper is particularly interesting. As most others do, the selected non–functional properties, reveal the physical footprint of a software system, that is a growing interest in the cyber–physical computing community.

The profiling procedure of the selected software components is consisted of the following steps:

1) *Initial measurement*, consisted of taking initial samples to get a notion about the characteristics of the measured non–functional properties for the selected computing units and components.
2) *Initial statistics*, involved verifying the standard deviation and the confidence interval of the initial data samples[11].
3) *Determining number of samples*, based on the initial measurements for the final measurement in order to satisfy the statistical confidence interval of 95% (95%CI).
4) *Performing the real measurement*, involved collecting the data which is used later on for the validation.
5) *Statistical analysis*, meant determining the standard deviation, the mean and performing the *t*–test.

In the next two subsections we further examine the measurement process for both the mean execution time and the average power consumption.

6.3. Measurement process for the mean execution time

For each Image filtering configuration the mean execution time was measured, recorded and analyzed. In this sub–section we elaborate the details of this procedure.

6.3.1. CPU

The mean execution time measurements of all components took place on the top most layer, i.e. within Java glue code.[12] The left side of the Fig. 2 shows the measurement points we used to obtain the mean execution time of components allocated on the CPU. As shown, we used A as a starting point and D as an ending point (in the code execution process), to measure the elapsed time as $D - A$.

6.3.2. GPU

The left part of the Fig. 2 also shows the mean execution time measurement points for the components allocated on the GPU. Points A and D were used as the starting and ending time for recording the total *end-to-end* elapsed time. This also includes the GPU kernel execution (real GPU processing time) given as $C - B$ and the data transfer time given as $(B - A) + (D - C)$. Since in reality, the GPU cannot perform data processing without a CPU, the *end-to-end* time was used in the final analysis.

6.3.3. FPGA

Similarly to the GPU, for the FPGA four measurement points were used as shown on the right side of the Fig. 2. The total *end-to-end* execution time is $D - A$, the FPGA execution time is obtained as $C - B$, and lastly, the data loading time is $(B - A) + (D - C)$.

To obtain reliable data, the measurements were repeated up to 10,000 times. The number of measurement repetitions was calculated with intention that all the collected data samples satisfy the statistical confidence interval of 95%.

6.4. Measurement process for the average power consumption

For power profiling of the software components we used a

[11] The analysis was performed with Minitab statistical tool, available at https://www.minitab.com/, accessed through the summer and winter of 2015.

[12] We used `System.nanoTime()`, however since it does not guarantee nanosecond accuracy, measurements were amended by the VisualVM tool found at http://visualvm.java.net/, accessed in February 2016.

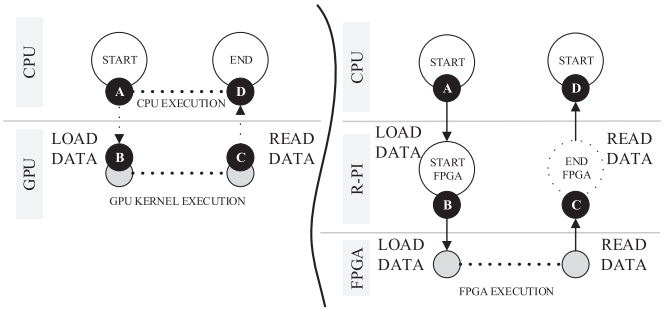

Fig. 2. Time measurement points A, B, C and D for all of the used computing units.

high–precision digital multimeter[13] which allows to simultaneously measure current and voltage, setting the sampling rate and storing the measurements to a USB device for later analysis. Initially, the Intels' Power Gadget tool was used along with the multimeter. However it proved to provide poor quality of results (varying up to ± 30%). Therefore, it was used to amend CPU monitoring data by `htop`, a Linux system–monitor. In the following text the details of the measurement process for each of the computing units are elaborated.

6.4.1. CPU

Since a CPU is connected to the motherboard through several hundreds of pins, it is inaccessible for the direct measurement of current. For that reason, the power was measured at the point *A* shown in Fig. 3.

Although it does provide power consumption for the entire motherboard, we used a technique inspired by Collange et.al. to obtain only the CPUs' power consumption which is further discussed in the next sub-section [30]. To use this method properly, it was first necessary to obtain the power consumption of the idling CPU [31].

The first measurements have shown that the current and voltage of the idling CPU were stable, so in order to obtain its average power consumption 4900 samples were necessary for both the current and the voltage, at a 10 Hz rate.

6.4.2. GPU

Initially the point *C* with a PCI-E pull–up board was used, however this exposed only the 3.3 V rail. The measurements have shown that 30–40% of the current flows through the 12 V rail which could not be accessed at the point C. Using current clamps was not an option due to their imprecision and poor sampling rate, so instead the current was measured at the point *A* using the same method as for the CPU.

The initial measurements of the idling average power consumption for the GPU have shown that for the 95%CI 4900 samples were necessary, taken at a 10 Hz rate.

6.4.3. FPGA

Using a modified USB cable which provided the access to the circuit series, average power consumption measurement of the FPGA board was performed at the point *B*. 1550 samples were taken at a 10Hz sampling rate, which was considerably less than for the CPU and the GPU due to the low variability of the collected data.

6.5. Trustworthiness of the collected data

Considering the average power consumption measurement points used for the component power profiling, one can argue that it is not possible to obtain 100% accurate data. Reasonably arguing that the recorded data for the individual computing units also contains the power consumption of the supporting hardware infrastructure, namely the motherboard, the memory and the storage. However, since this infrastructure is essential and the computing units cannot be really decoupled from the computing system. To solve this issue, we used an approach inspired by the work of Collange et.al. [30]. In our case, one does not really need to acquire the accurate average power consumption data for each computing unit. Instead, it is necessary to obtain the accurate change of the measured average power consumption while the system is subjected to the intentional processing load, i.e. component execution.

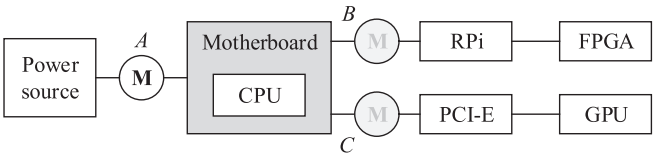

Fig. 3. Power consumption measurement points.

To achieve this, one needs to obtain the idling average power consumption of each computing unit, measured with next to zero processing load [31,32]. Now, having the average power consumption of the idling unit any execution of software components, i.e. data processing causes the average power consumption to increase above this idling value. It is this increase, that makes it possible to make a distinction of the influence that a software component has on the average power consumption of a computing unit.

Fig. 4 shows the physical footprint created by the software on the current consumption curve. The grayed out area contains the usable data, while the remaining data represents intentionally added and controlled delay between component function calls for two reasons. First, for easier detection in later analysis, and second to allow the measured values to settle.

The same approach was used for both the CPU and the GPU average power consumption. Subtracting the idling CPU average power consumption from the idling GPU average power consumption, one can obtain the average power consumption of the GPU alone, and then detect changes. Furthermore, with this method one can monitor the true average power consumption of the GPU components (with the indirect load it imposes to the CPU) and ultimately, in this way one can compare average power consumption of various software configurations.

6.6. The results

Applying the previously described measurement process, the measurements resulted with two datasets. One related to the average power consumption of the idle HCP and the other related to the mean execution time and average power consumption for the Image filtering and Object detection components.

6.7. Average power consumption of the idle heterogeneous computing platform

Table 1 shows the average power consumption of the heterogeneous computing platform without the processing load (Mean P), the of usable samples (N), the standard deviation (std.dev) and the upper and lower bound of the confidence interval (95% CI). As discussed above, the main interest is the change of these values as a consequence of components executing tasks.

With only the CPU, the idle system consumes 9.23 W. Plugging in the GPU, this value increases to the 12.12 W, meaning that the GPU increases the average power consumption for 2.89 W (a low-profile GPU was used). Introducing a GPU to the system reduced the CPUs load for 15%, which were responsible for handling graphics related tasks. However, in reality the GPU consumes a bit more than 2.89 W, but this

[13] GwInstek GDM–8342, Codegen 400W power supply, model 300XX.

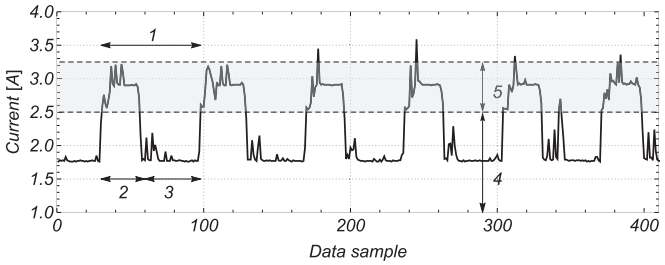

Fig. 4. Measurement sample, 1 – function call, 2 – GPU active, 3 – intentional idle, 4 – discarded data, 5 – area with real average power consumption (gray box).

Table 1
Average power consumption of the idling CPU, GPU and FPGA.

Variable	N	Mean P(W)	Std. dev.	95% CI (upper and lower)	
CPU	4151	9.2393	0.0855	9.2367	9.2419
CPU and GPU	4695	12.122	0.0730	12.1204	12.1246
FPGA	1550	2.3183	0.0294	2.3168	2.3197

is of no concern, since the only interest here is in the change of this value, for various software configurations. The idling FPGA uses 2.31 W.

6.8. Measurement results

6.8.1. Image filtering component

Table 2 shows the average power consumption and the mean execution time for the Image filtering component. To interpret the table, consider the row *SG* for the average power consumption, and the column *QVGA* which has the value 3.0382. This number means that the average power consumption of the CPU increases by 3.0382 W above the idling average power consumption, while it hosts the *Sobel* and the *Gauss* sub–components, i.e. filters and executes them in the *pipe&filter* architectural style.

A closer look into the table reveals that the increase of components, i.e. filtering operations does not increase the average power consumption as much as the input image size does. Also, it is noticeable how the average power consumption for some computing units, which host multiple components, does not increase proportionally. It is due to the aforementioned synergy effect, apparent throughout the table (e.g. compare the QVGA and VGA rows for the S and G alone to the S, G operations together).

And while the average power consumption for the CPU and GPU changes significantly, for the FPGA this is not the case, regardless to the number of hosted components. Similar observations were made by Flowers [33] and Kestur [34], which occur because an FPGA intellectual property core tends to use the same number of logic blocks regardless to the number of used operations.

For the mean execution time, the same observations can be noticed as for the average power consumption.

6.8.2. Object detection component

The results of measuring the average power consumption for the Object detection component are shown in Table 3. The number 2.5818 in the VGA row II, for the CPU average power consumption column is the average increase of the average power consumption in watts, in the case where the Object detection component allocated to the CPU is processing the second VGA image (marked with II).

As it can be seen, the CPU was more power efficient while the GPU was more time efficient. On average, the CPU consumed 39% less power than the GPU, however, on average it needed 74% more time to process the images.

6.9. The conclusion of the measurements

The Fig. 5 (a) shows the mean difference of energy[14] between the GPU and the CPU required by the Image filtering component.

The (b) part shows the same data, but for the Object detection component. Redder areas signify the bigger difference, meaning that the GPU required less energy, while the bluer areas signify less difference meaning that the CPU required less energy. In most cases the GPU seems to be more efficient. However, there is a blue cluster which suggests that for certain components and their inputs the CPU is potentially more energy efficient.

Not only does this show that with two non–functional properties it is hard enough to make design decisions in heterogeneous computing, but also that choosing the component deployment design is neither intuitive nor straightforward. In addition, a system designer also needs to consider different data inputs for the available components, i.e. *usage scenarios* apparent through component input data, which clearly influence the performance of an entire heterogeneous computing system.

7. Framework validation – demonstrator

To prove that SCAF model can produce the optimal software configuration outside the theoretical boundaries within which it is currently presented, this section presents its empirical validation on a real–world heterogeneous computing system.

The validation of SCAF is performed by extensive and thorough experimentation on the formerly introduced heterogeneous computing system, the purpose of which is to prove that SCAF correctly represents the behavior of a real–world heterogeneous computing system. The proposed experiment involves the following steps:

1. Define the heterogeneous computing system and design the software configuration: (a) by informed (using previous measurements) manual component allocation using the results previous measurements in order to (1) set a benchmark for SCAF and (2) to validate if SCAF provides better configurations, (b) by applying SCAF.
2. Using the cost function *w* quantify the performance of manually defined and SCAF produced configurations and rank the configurations by the result (Rank 1).
3. Implement the manually defined and SCAF produced configurations on a real–world heterogeneous computing system and measure their performance. Rank the allocations in accordance to the results of measurements (Rank 2).
4. Compare the Rank 1 and 2. If the ranks match, the cost function *w* correctly represents the real–world heterogeneous computing system behavior. If the ranks do not match, cost function *w* needs to be rejected.

7.1. The experiment

For the experiment we used the heterogeneous computing system described in Section 6.1.1.

7.1.1. Determining the experimental software configurations

In accordance to the remarks of the previous chapter related to input parameters of software components and their impact on the configuration performance, we defined two execution scenarios. Each scenario is defined by a set of sequentially executed software components. Each element of this scenario is a standalone software component performing a certain set of image filters. The two scenarios differ by the processing resource demand; they are:

Scenario 1: $S_{VGA} \rightarrow SGE_{VGA} \rightarrow SGE_{SXGA} \rightarrow SGEDH_{SXGA} \rightarrow OD_{VGA} \rightarrow OD_{FHD} \rightarrow SGED_{FHD} \rightarrow SGEDH_{FHD} \rightarrow S_{QVGA} \rightarrow G_{QVGA} \rightarrow$

[14] Energy addresses both time and power.

Table 2
Image filtering component results for different inputs and operations. Rows show the computing platform and the input image size, while the columns display software component filters.

	CPU					GPU					FPGA
Resolution	QVGA	VGA	SXGA	FHD	8KFD	QVGA	VGA	SXGA	FHD	8KFD	QVGA
					Average execution time in milliseconds						
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
SG	3.11	8.36	31.74	52.80	1590.83	0.83	3.46	13.67	17.43	289.80	125.10
SGE	3.32	10.29	37.55	67.07	1909.40	1.16	5.00	14.38	16.87	380.75	125.47
SGED	3.63	11.11	49.26	76.24	2235.89	1.59	7.00	15.48	19.39	529.36	124.87
SGEDH	4.04	11.47	50.04	80.43	2345.60	1.49	5.80	15.12	18.22	481.97	124.62
					Average power consumption in Watts						
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
SG	3.0382	3.2566	3.9846	7.2490	16.2969	5.8686	5.6862	6.7314	9.4641	20.5100	2.4348
SGE	3.1493	3.5405	4.4101	8.2373	16.4579	6.0381	5.9074	9.6826	9.9184	20.6782	2.4352
SGED	3.3123	3.6256	7.5179	9.2339	16.4832	6.1793	6.2664	9.9228	10.6593	20.6151	2.4318
SGEDH	3.3966	3.8535	7.6358	9.1231	16.7893	6.2688	6.4524	9.9566	11.4692	21.3956	2.4383

Table 3
Object detection component, average power consumption and mean execution time, 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
			Average power consumption in Watts			
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
			Average execution time in milliseconds			
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

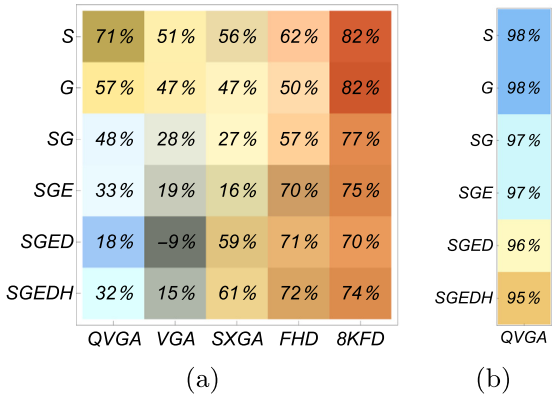

Fig. 5. Difference in energy requirement for the Image filtering (a) and Object detection (b) components for the CPU and GPU. Red means less energy is required by the GPU while blue means less energy required by the CPU. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

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 arrow symbol determines the order of software component execution (one after another), while the component is named by the image size it can process and the image filters it executes. E.g. "$SGE_{VGA} \rightarrow SGE_{SXGA}$" indicates the execution of *SGE* filter on a *VGA* image, followed by the execution of *SGE* filter on *SXGA* image. Although the operation is the same, the components can have only one image size as a parameter. This is why these are built as two separate components, in order to increase the experimental design space. These two scenarios are now to be allocated to computing units.

In order to establish a benchmark for SCAF and verify whether it produces a better configuration than a software architect would manually, using the results of measurements shown in Section 6 we defined two manual configurations by intuitive placement[15]. These are shown in Table 4.

Using the measurement results presented in earlier sections, the software configuration of the demonstrator platform was optimized using SCAF. For both scenarios, Table 5 shows configurations produced by SCAF optimization. For the first scenario, majority of the of the components, 8 out of 11, were allocated to the CPU. The remaining 3 were allocated on the GPU, and all of which have the FHD input images. No components were allocated to the FPGA.

The previous software configuration optimization does not apply the Synergy effect approximation matrix $\mathcal{S}$. However, by defining it and applying it with SCAF, the configurations change. Table 6 shows that for the first scenario, the load between the CPU and the GPU was balanced out, and no components were allocated to the FPGA. In the second scenario, the FPGA was used for the components with QVGA images, which reduced the CPUs load.

7.1.2. Ranking the configurations by performance (obtaining Rank 1 and Rank 2)

Allocating software components manually and by SCAF produced the total of six allocation vectors, i.e. software configurations shown in Table 7.

Each configuration $\alpha^{(1-6)}$ was assessed by the SCAF cost function w. Table 8 shows the result of ranking the configurations by their performance, i.e. the prediction of their performance in the real world (lower is better). The trade–off vector $\mathcal{F}$ for only two values is trivial. We have chosen for the average power consumption to have two times larger significance over the mean execution time for our scenarios, therefore the weights are 0.66 for the former and 0.33 for the latter.

Table 9 shows the ranking of configurations $\alpha^{(1-6)}$ by their performance measured on a real–world platform.

The most important column is the rank which shows the exact match of the two previous rankings the one given by w shown in Table 8 and the one given by measurements shown in Table 9. This means that our modeling framework correctly represents the heterogeneous

[15] Intuitive placement refers to an assumption that each component should be placed to a computing unit for which it prooved to be efficient in the previous evaluation, shown in the Section 6.6. The CPU was efficient for QVGA and SXGA, the GPU for FHD and the FPGA for QVGA.

Table 4
Manually defined configurations.

Computing unit	Components
	Scenario 1
CPU	S_{VGA}, SGE_{VGA}, SGE_{SXGA}, $SGEDH_{SXGA}$
GPU	OD_{VGA}, OD_{FHD}, $SGED_{FHD}$, $SGEDH_{FHD}$
FPGA	S_{QVGA}, G_{QVGA}, SG_{QVGA}
	Scenario 2
CPU	SG_{SXGA}, SGE_{SXGA}
GPU	S_{FHD}, G_{FHD}, SG_{FHD}, SGE_{FHD}, $SGEDH_{FHD}$, OD_{FHD}
FPGA	S_{QVGA}, SGE_{QVGA}, $SGEDH_{QVGA}$

Table 5
Configurations produced by SCAF.

Computing unit	Components
	Scenario 1
CPU	S_{VGA}, SGE_{VGA}, SGE_{SXGA}, $SGEDH_{SXGA}$, OD_{VGA}, S_{QVGA}, G_{QVGA}, SG_{QVGA}
GPU	OD_{FHD}, $SGED_{FHD}$, $SGEDH_{FHD}$
FPGA	—
	Scenario 2
CPU	SG_{SXGA}, SGE_{SXGA}, G_{FHD}, SG_{FHD}, S_{QVGA}, SGE_{QVGA}, $SGEDH_{QVGA}$
GPU	S_{FHD}, SGE_{FHD}, $SGEDH_{FHD}$, OD_{FHD}
FPGA	—

Table 6
SCAF configurations, with synergy tradeoff applied.

Computing unit	Components
	Scenario 1
CPU	SGE_{VGA}, SGE_{SXGA}, OD_{VGA}, S_{QVGA}, G_{QVGA}, SG_{QVGA}
GPU	S_{VGA}, OD_{FHD}, $SGEDH_{SXGA}$, $SGED_{FHD}$, $SGEDH_{FHD}$
FPGA	—
	Scenario 2
CPU	SG_{SXGA}, SGE_{SXGA}, G_{FHD}, S_{FHD}, S_{QVGA}
GPU	SG_{FHD}, SGE_{FHD}, $SGEDH_{FHD}$, OD_{FHD}
FPGA	SGE_{QVGA}, $SGEDH_{QVGA}$

Table 7
The implemented experimental configurations.

Configuration	Scenario	Produced with
$\alpha^{(1)}$	1	SCAF
$\alpha^{(2)}$	1	SCAF + $\mathcal{S}$
$\alpha^{(3)}$	1	manually
$\alpha^{(4)}$	2	SCAF
$\alpha^{(5)}$	2	SCAF + $\mathcal{S}$
$\alpha^{(6)}$	2	manually

Table 8
Ranking of the experimental configurations by the cost function w (Rank 1).

Allocation	Evaluation result (w)	Rank
	Without $\mathcal{S}$	
$\alpha^{(1)}$	2,6957	1
$\alpha^{(4)}$	2,9374	2
$\alpha^{(3)}$	3,3888	3
$\alpha^{(6)}$	3,5908	4
	With $\mathcal{S}$	
$\alpha^{(2)}$	1,7155	1
$\alpha^{(5)}$	1,8626	2
$\alpha^{(3)}$	1,9348	3
$\alpha^{(6)}$	2,2326	4

computing platform, and that it is capable of correctly predicting the performance of different software configurations. Notice that with the application of the Synergy effect matrix $\mathcal{S}$ the result ($w(\alpha^{(2)}) = 1447.48$) was better than in the case without it ($w(\alpha^{(1)}) = 1514.07$), which further enhances the validity of our modeling framework.

Table 9
Comparison of manually defined configurations and SCAF produced configurations (Rank 2).

Allocation	APC (W)	MET (ms)	Result	Rank
		Without $\mathcal{S}$		
$\alpha^{(1)}$	323924	4153,5888	1514,0796	1
$\alpha^{(4)}$	32,1609	6221,8502	2258,5194	2
$\alpha^{(3)}$	31,0731	8887,6262	3217,5678	3
$\alpha^{(6)}$	31,9835	11071,8959	4004,4329	4
		With $\mathcal{S}$		
$\alpha^{(2)}$	32,9684	4051,0150	1477,4871	1
$\alpha^{(5)}$	33,2779	6226,1707	2260,7226	2
$\alpha^{(3)}$	31,0731	8887,6262	3217,5678	3
$\alpha^{(6)}$	31,9835	11071,8959	4004,4329	4

Table 10
Simulated annealing - Simannel parameters.

Parameter	Value	Interpretation
T_{max}	1,000,000	Starting temperature
T_{min}	0.5	Ending temperature
i	20,000	Number of iterations

Table 11
Genetic algorithm - DEAP parameters.

Param.	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.

8. SCAF for large design spaces

Having several thousands of allocation options, the design space for the presented heterogeneous computing system is relatively small, and in such cases the optimal configuration can be found by using the exhaustive search method. However, with the increase of components and computing units the design space rapidly grows and soon enough the exhaustive search is infeasible. This section introduces two heuristic methods; the genetic algorithm and the simulated annealing which were used to generate sufficient sub–optimal configurations.

To verify the quality of the configurations generated by the selected heuristic methods, two experiments were conducted. The first one dealing with the average difference in performance between the optimal and sub–optimal configurations, which is for smaller design spaces; and the second one dealing with comparing configurations generated by heuristic methods in comparison to randomly generated configurations, which is for very large design spaces. Having said that, we need to address the issue of benchmarking software configurations within this context. To the best of our knowledge, there are no standardized methods by which researchers and practitioners can perform evaluation and comparison of their methods of optimizing software architecture. While investigating on this issue, we found that many authors appeal to the same issue [31,35–37], which should be addressed in future.

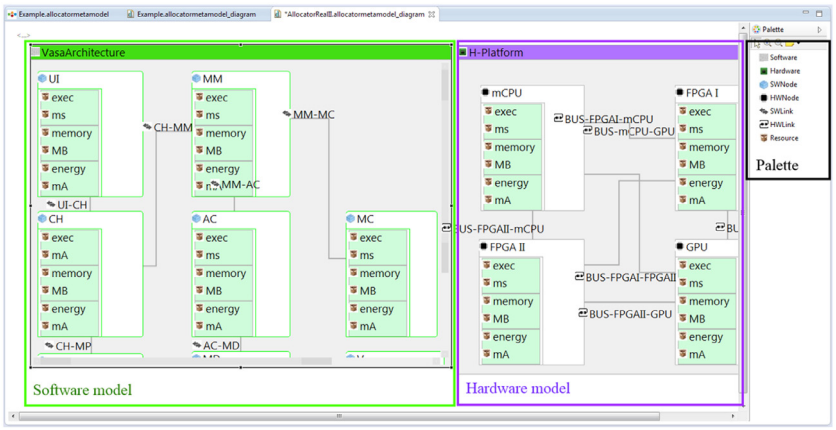

Fig. 6. SCALL screenshot showing hardware and software models side–by–side, i.e. a heterogeneous computing system.

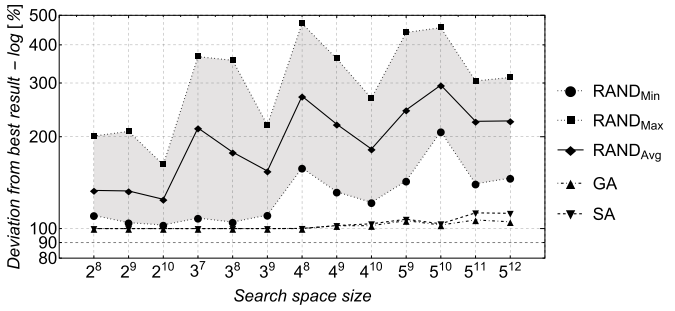

Fig. 7. Difference in percentage between allocation performance obtained by the GA, SA and RAND (min, max, avg).

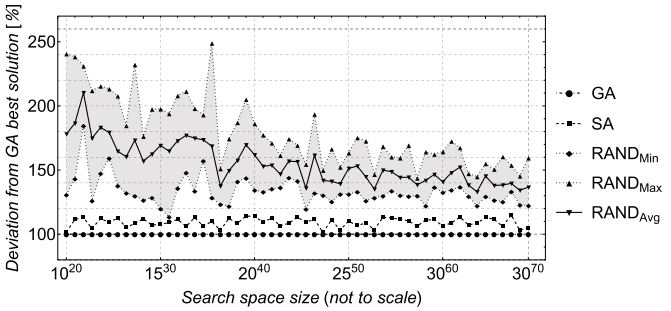

Fig. 8. Difference in allocation performance as a percentage offset from the GA solution.

To mitigate this issue, we randomly generated the parameters for 55 different software architectures and compared SCAF against them, using the central limit theorem and the rigorous standard of confidence interval of 95%. With this approach, SCAF consistently produced significantly better and relevant software configurations, than randomly generated ones. To clarify further, in order to optimize the 55 randomly generated software architectures and benchmark the performance of the results we used four different approaches: a) SCAF using the genetic algorithm (GA), b) SCAF using the simulated annealing (SA), c) performing an exhaustive search (ES) and d) by choosing the best, the worst and the average performing configuration out of 30 consecutive random configurations (RAND). Due to the lack of benchmarking methods and frameworks, we used RAND to prove that SCAF provides better result than a simple random guess, which proved to not always be the case [38].

8.1. Optimization implementation

SCAF does not explicitly define the optimization method, however we are using the GA and SA method since they performed well in our preliminary study. In addition, other authors report to obtain satisfactory results for assignment and scheduling problems [39–41]. Their further examination is out of scope this paper.

Implementation and setup of the Simulated annealing – for the implementation of SCAF using the Simulated annealing optimization we used the Simanneal library[16] for Python 3.5. The parameters and the setup used for the experiments are shown in Table 10.

Implementation and setup of the Genetic algorithm – we used Python 3.5 and DEAP library [42]. Table 11 shows its setup and parameters.

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

8.2. SCAF implementation

SCAF is implemented in as a part of Software Allocator (SCALL) tool [43], our custom Eclipse plug–in which consists of two main parts:

a) *Eclipse based model editor* – is used to design the heterogeneous computing system and represent it using a standard EMF Ecore notation corresponding to SCAF as shown in Fig. 6,
b) *PyAllocator* – a Python script which employs our multi–objective heuristic allocation method.

SCALL currently provides the following features; (a) *model creation* – the user can simultaneously model software and hardware architecture of the system, (b) *model visualization* – allowing easier model creation, manipulation and overview, (c) *pairwise resource comparison* – for the aforementioned AHP process simplification, and (d) *software configuration optimization* – by employing multi–objective software component allocation.

8.3. Experiment 1: Exhaustive search vs. heuristic methods

The goal of this experiment is to determine how much does the performance of configurations generated by the GA, SA and RAND differ from the performance of the optimal configuration found by the exhaustive search. In order to accomplish this, the exhaustive search was performed once in order to find the real optimal configuration, while the remaining methods were repeated 30 times. The input parameters for the SCAF were randomly generated[17] for design spaces ranging between 2^8 and 5^{12}. The performances of the configurations

[17] The algorithms are implemented in Python 3.5 and the calculations were made on a server with 2 × Intel®Xeon®CPU E7-4830, 8GB RAM, and Linux.

obtained by each approach are compared in Fig. 7. It shows that the configurations provided by the GA and SA closely follow the optimal configuration obtained by ES, and also it shows that randomly obtained allocations provide a variety of different performing configurations, degrading even more with increase of design space.

Considering the processing time and the performance of the generated configurations, the GA was the best. On average, for all the observed design spaces the time to obtain the sub-optimal configuration for the GA approach was less than 10 s and the SA took between 10 and 21 s. The processing time for the ES is hardly comparable as it took more than 1 day and 14 h for the largest design space. The RAND was not considered since it is pure random number generation lasting just a fraction of a second.

As for the performance, the worst performing configurations generated by the GA and SA were respectively 7% and 13% worse. The configurations provided by the RAND for smaller design spaces were at best 3% worse than the optimal ones, but for larger ones this increased up to 22% worse at best, and 100% worse on average.

The configurations generated by the GA and SA provide the acceptable trade-off between the performance and the time necessary to obtain it. As such, these methods were adopted as acceptable for design spaces for which the exhaustive search is not feasible.

8.4. Experiment 2: Heuristic methods vs. randomized method

The goal of the second experiment is to examine how do configurations provided by the heuristic methods compare to randomly generated ones. For this experiment, 55 heterogeneous computing system models were randomly generated with design spaces ranging between 10^{20} and 30^{70}. The performance of the generated configurations (given by w) for each of the remaining methods are shown in Fig. 8. It can be noticed that the GA and the SA approach provide the best performing configurations. For smaller design spaces RAND provides configurations with a large diversity of performance, which decreases with the increase of the design space. However, not in a single case do configurations generated by RAND reach the best performing configurations given by GA or SA. At best RAND configurations are on average 33% worse than those generated by GA, and 22% worse than the ones generated by SA. On average, RAND configurations are 55% worse than GA and 42% then SA. Therefore, it can be concluded that the configurations generated by the SCAF have superior performance than any random configuration does, in a significant number of different cases.

9. Conclusion

Choosing the best software configuration for heterogeneous computing systems is not a straightforward task as it involves multi-dimensional decision making with multiple non–functional properties of different importance, used through various execution scenarios and an exponentially growing design space. Software architects currently lack the models and automated tools to simplify software architecture optimization for heterogeneous computing systems.

Having performed a thorough investigation of this issue, our research resulted with SCAF, a software allocation framework which fuses all the necessary data about a heterogeneous computing system through a unified formal model in order to provide the (sub–) optimal software configuration and to predict the performance of a system in its earliest design pase. We performed rigors measurements (within the confidence interval of 95%) and extensive experiments to validate SCAF in the real world, while focusing on multiple non–functional properties, or in particular the mean execution time and the average power consumption. These were performed on a custom built heterogeneous computing system, containing a CPU, a GPU and an FPGA, along with a software system composed of more than 30 components. Not only did SCAF prove to provide sufficient sub–optimal software configurations in comparison to the exhaustive search methods, but it also proved to correctly predict the performance of a HCP. Using the model of a heterogeneous computing system with a very large design space, our implementation of SCAF provides superior configurations in a significant number of cases than the best randomly generated configurations.

The thorough and rigorous measurements performed for data collection used by our model might be unpractical for frequent use by system architects. We would like to point out that we resorted to this means to validate, increase confidence in SCAF and to prove a concept. However, there are other methods of data collection for the proposed model, e.g. performing less complex measurements, using software component and computing unit specifications or estimations based on experiences (which is common in the industry).

And while SCAF offers advantages to current software configuration design decision methods, we found some interesting research subjects which are under-explored: (a) analysis of heuristic algorithms to further enhance SCAF configurations, (b) addressing run–time system dynamics and communication, (c) experiment with SCAF with systems concerned with software configurations issues, beyond robotics, e.g. docker allocation and hardware–software co–design.

References

[1] W. Afzal, R. Torkar, R. Feldt, A systematic review of search-based testing for non-functional system properties, Inf. Softw. Technol. 51 (6) (2009) 957–976.

[2] S. Rekha, H. Muccini, Group decision-making in software architecture: a study on industrial practices, Inf. Softw. Technol. (2018).

[3] W. Wolf, Cyber-physical systems, IEEE Comput. 42 (3) (2009) 88–89, https://doi.org/10.1109/MC.2009.81.

[4] R. Poovendran, Cyber-physical systems: close encounters between two parallel worlds, Proc. IEEE 98 (8) (2010) 1363–1366, https://doi.org/10.1109/JPROC.2010.2050377.

[5] I. Crnkovic, M.P.H. Larsson, Building Reliable Component-Based Software Systems, Artech House, 2002.

[6] I. Crnkovic, S. Sentilles, A. Vulgarakis, M.R.V. Chaudron, A classification framework for software component models, Softw. Eng. IEEE Trans. 37 (5) (2011) 593–615, https://doi.org/10.1109/TSE.2010.83.

[7] S.H. Choi, I.B. Jeong, J.H. Kim, J.J. Lee, Context generator and behavior translator in a multilayer architecture for a modular development process of cyber-physical robot systems, IEEE Trans. Ind. Electron. 61 (2) (2014) 882–892, https://doi.org/10.1109/TIE.2013.2254095.

[8] T. Vale, I. Crnkovic, E.S. de Almeida, P.A.d.M.S. Neto, Y.C. Cavalcanti, S.R. de Lemos Meira, Twenty-eight years of component-based software engineering, J. Syst. Softw. 111 (2016) 128–148, https://doi.org/10.1016/j.jss.2015.09.019.

[9] A. Koziolek, 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, (2011), pp. 103–108, https://doi.org/10.1145/2000229.2000244.

[10] S. Malek, N. Medvidovic, M. Mikic-Rakic, An extensible framework for improving a distributed software system's deployment architecture, IEEE Trans. Software Eng. 38 (1) (2012) 73–100.

[11] A.D. Pimentel, C. Erbas, S. Polstra, A systematic approach to exploring embedded system architectures at multiple abstraction levels, IEEE Trans. Comput. 55 (2) (2006) 99–111, https://doi.org/10.1109/TC.2006.16.

[12] A. Martens, D. Ardagna, H. Koziolek, R. Mirandola, 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 (2010), p. 84, https://doi.org/10.1007/978-3-642-13821-8.

[13] S. Becker, H. Koziolek, R. Reussner, The palladio component model for model-driven performance prediction, J. Syst. Softw. 82 (1) (2009) 3–22, https://doi.org/10.1016/j.jss.2008.03.066.

[14] A. Martens, H. Koziolek, S. Becker, 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, (2010), pp. 105–116, https://doi.org/10.1145/1712605.1712624.

[15] S. Islam, R. Lindström, N. Suri, Dependability driven integration of mixed criticality SW components, Proceedings - Ninth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, ISORC 2006, (2006), pp. 485–495. doi:10.1109/ISORC.2006 .

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

[17] A.K. Singh, M. Shafique, A. Kumar, J. Henkel, Mapping on multi/many-core systems: survey of current and emerging trends, Proceedings of the 50th Annual Design Automation Conference, ACM, 2013, p. 1.

[18] S. Murali, M. Coenen, A. Radulescu, K. Goossens, G. De Micheli, A methodology for mapping multiple use-cases onto networks on chips, Proceedings of the conference

on Design, automation and test in Europe: Proceedings, European Design and Automation Association, 2006, pp. 118–123.
[19] C.-E. Rhee, H.-Y. Jeong, S. Ha, Many-to-many core-switch mapping in 2-d mesh noc architectures, Computer Design: VLSI in Computers and Processors, 2004. ICCD 2004. Proceedings. IEEE International Conference on, IEEE, 2004, pp. 438–443.
[20] J. Hu, R. Marculescu, Energy-and performance-aware mapping for regular noc architectures, IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. 24 (4) (2005) 551–562.
[21] C.A.M. Marcon, E.I. Moreno, N.L.V. Calazans, F.G. Moraes, Comparison of network-on-chip mapping algorithms targeting low energy consumption, IET Comput. Digital Techniq. 2 (6) (2008) 471–482.
[22] G. Ascia, V. Catania, M. Palesi, Multi-objective mapping for mesh-based noc architectures, Proceedings of the 2nd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, ACM, 2004, pp. 182–187.
[23] R. Leupers, L. Thiele, A.A. Jerraya, P. Vicini, P.S. Paolucci, Shapes:: a tiled scalable software hardware architecture platform for embedded systems, Hardware/Software Codesign and System Synthesis, 2006. CODES+ ISSS'06. Proceedings of the 4th International Conference, IEEE, 2006, pp. 167–172.
[24] L. Thiele, I. Bacivarov, W. Haid, K. Huang, Mapping applications to tiled multiprocessor embedded systems, Application of Concurrency to System Design, 2007. ACSD 2007. Seventh International Conference on, IEEE, 2007, pp. 29–40.
[25] A. Aleti, B. Buhnova, L. Grunske, A. Koziolek, I. Meedeniya, Software architecture optimization methods: a systematic literature review, IEEE Trans. Software Eng. 39 (5) (2013) 658–683, https://doi.org/10.1109/TSE.2012.64.
[26] J. Hartmanis, Computers and intractability: a guide to the theory of np-completeness (michael r. garey and david s. johnson), SIAM Rev. 24 (1) (1982) 90.
[27] W. Wolf, Computers as Components, 1 edition, Elsevier, 2012, https://doi.org/10.1016/B978-0-12-388436-7.00004-0.
[28] P. Marwedel, Embedded system design: Embedded systems foundations of cyber-Physical systems, (2010).
[29] T. Saaty, Fundamentals of decision making and prority theory with the analytic hierarchy process, RWS Publications, 1994.
[30] S. Collange, D. Defour, A. Tisserand, Power consumption of GPUs from a software perspective, Lect. Notes Comput. Sci. 5544 LNCS (PART 1) (2009) 914–923, https://doi.org/10.1007/978-3-642-01970-8_92.
[31] K. Grosskop, J. Visser, Energy efficiency optimization of application software, Advances in Computers, vol. 88, Elsevier, 2013, pp. 199–241.
[32] I. Švogor, An initial performance review of software components for a heterogeneous computing platform, Proceedings of the 2015 European Conference on Software Architecture Workshops, ECSAW '15, ACM, New York, NY, USA, 2015, pp. 16:1–16:4, https://doi.org/10.1145/2797433.2797449.
[33] J. Fowers, G. Brown, P. Cooke, 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, (2012), p. 47, https://doi.org/10.1145/2145694.2145704.
[34] S. Kestur, J.D. Davis, O. Williams, BLAS Comparison on FPGA,CPU and GPU, Proceedings - IEEE Annual Symposium on VLSI, ISVLSI 2010 (2010), (2010), pp. 288–293. doi:10.1109ASVLSI.2010.84 .
[35] A.E. Trefethen, J. Thiyagalingam, Energy-aware software: challenges, opportunities and strategies, J. Comput. Sci. 4 (6) (2013) 444–449.
[36] E. Jagroep, A. van der Ent, J.M.E. van der Werf, J. Hage, L. Blom, R. van Vliet, S. Brinkkemper, The hunt for the guzzler: architecture-based energy profiling using stubs, Inf. Softw. Technol. (2017).
[37] C.C. Venters, R. Capilla, S. Betz, B. Penzenstadler, T. Crick, S. Crouch, E.Y. Nakagawa, C. Becker, C. Carrillo, Software sustainability: research and practice from a software architecture viewpoint, J. Syst. Softw. (2017).
[38] J. Feljan, J. Carlson, Task allocation optimization for multicore embedded systems, Proceedings - 40th Euromicro Conference Series on Software Engineering and Advanced Applications, SEAA 2014, (2014), pp. 237–244, https://doi.org/10.1109/SEAA.2014.22.
[39] G.G. Pascual, R.E. Lopez-Herrejon, M. Pinto, L. Fuentes, A. Egyed, Applying multiobjective evolutionary algorithms to dynamic software product lines for reconfiguring mobile applications, J. Syst. Softw. 103 (2015) 392–411, https://doi.org/10.1016/j.jss.2014.12.041.
[40] Y.-K. Kwok, I. Ahmad, Static scheduling algorithms for allocating directed task graphs to multiprocessors, ACM Comput. Surv. 31 (4) (1999) 406–471, https://doi.org/10.1145/344588.344618.
[41] A. Ramírez, J.R. Romero, S. Ventura, An approach for the evolutionary discovery of software architectures, Inf. Sci. 305 (2015) 234–255, https://doi.org/10.1016/j.ins.2015.01.017.
[42] F.-A. Fortunm, F.-M. De Rainville, M.-A. Gardner, M. Parizeau, C. Gagn, DEAP: Evolutionary algorithms made easy, J. Mach. Learn. Res. 13 (2012) 2171–2175. doi:10.1.1.413.6512 .
[43] I. Švogor, J. Carlson, Scall: Software component allocator for heterogeneous embedded systems, Proceedings of the 2015 European Conference on Software Architecture Workshops, ACM, 2015, p. 66, https://doi.org/10.1145/2797433.2797501.