Component Allocation Optimization for Heterogeneous CPU-GPU Embedded Systems

Gabriel Campeanu, Jan Carlson and Séverine Sentilles
Mälardalen Real-Time Research Center, Mälardalen University, Västerås, Sweden
Email: {gabriel.campeanu, jan.carlson, severine.sentilles}@mdh.se

***Abstract*—In a quest to improve system performance, embedded systems are today increasingly relying on heterogeneous platforms that combine different types of processing units such as CPUs, GPUs and FPGAs. However, having better hardware capability alone does not guarantee higher performance; how functionality is allocated onto the appropriate processing units strongly impacts the system performance as well. Yet, with this increase in hardware complexity, finding suitable allocation schemes is becoming a challenge as many new constraints and requirements must now be taken into consideration. In this paper, we present a formal model for allocation optimization of embedded systems which contains a mix of CPU and GPU processing nodes. The allocation takes into consideration the software and hardware architectures, the system requirements and criteria upon which the allocation should be optimized. In its current version, optimized allocation schemes are generated through an integer programming technique to balance the system resource utilization and to optimize the system performance using the GPU resources.**

I. Introduction

The rapid advances in microprocessor technology has favored the evolution of embedded systems, from homogeneous single core CPU systems to heterogeneous systems with multiple processors of different types (e.g., GPU, FPGA). Using FPGA nodes or collocating multicore CPUs with GPUs onto single nodes may increase the overall system performance by allowing distributing functionality to nodes with appropriate computation specializations.

While initially GPUs were used mostly for graphic-based applications, their increased computing power made researchers look at ways to utilize them in new contexts. For example, in the automotive industry, several research initiatives focus on using GPUs to implement vision systems for vehicles [7]. Other examples of GPU-based applications include autonomous vision-based robots [12], 3D reconstruction medical systems [20], etc. This development comes from the considerable speed-ups that can be achieved through GPUs compared to cases using CPUs. For example for the n-body simulation, 200x speed-up can be realized [15]. The GPU induced speed-up not only depends on the system hardware but also on how the functionality accesses the resources, such as the number of registers per thread, local, shared or global memory. Dividing the software application and running it on the appropriate computation node can be beneficial to improve the performance of the system, but also a higher complexity cost concerning the allocation process [6].

Determining which functionality should be placed on a given computation node is an NP-hard problem known as software deployment [5]. Unconstrained, the solution space grows exponentially with the number of nodes and software units. From the total solution space, only a subset are feasible deployment schemes, i.e. solutions that satisfy both the system requirements and the constraints in terms of hardware, functional and non-functional properties. The allocation problem becomes even harder when, instead of having only CPU-based hardware platforms, we use complex CPU-GPU based systems. In addition to the deployment constraints from the CPU nodes such as RAM memory usage or computation load, one must consider extra constraints from the GPU part, such as number of threads, registers per thread or shared memory usage. Distributing functionality over CPU-GPU nodes greatly influences the outcome of the system performance, and thus deciding the distribution of the hardware resources to the software application should be carefully analyzed.

This paper presents a software component allocation model for heterogeneous embedded systems composed of CPUs and GPUs. The main contribution of the paper resides in the formal description of the allocation model, including the software and hardware models. The software model contains components with platform-independent properties and the hardware model described the physical platform intended for deployment. The mathematical model for component allocation includes different constraints (CPU and GPU load and memory usage, etc.) and several optimization criteria (performance optimization, balancing the memory usage, etc.). To get a better scalability of the approach, the formal model only consider details of the models that are relevant for the allocation; other factors that may affect CPUs and GPUs performance (e.g., virtual memory, registers per thread, usage of shared memory) are abstracted away in the current version of work. For the evaluation part, we use an integer programming method [14] to generate deployment schemes. Expressing the formal allocation model into a nonlinear integer programming problem allows for different formal constraints to be easily integrated in the model, to meet various application requirements.

The remainder of the paper is structured as follows: Section II gives an overview of the approach with details on the software, hardware and allocation models. Section III introduces the mathematical formalization of the allocation model. Section IV presents the translation of the mathematical optimization model to a solver, followed by the evaluation of the model in Section V. Section VI discusses the contributions in relation to other research works and Section VII summarizes the work and opens up to opportunities for future work.

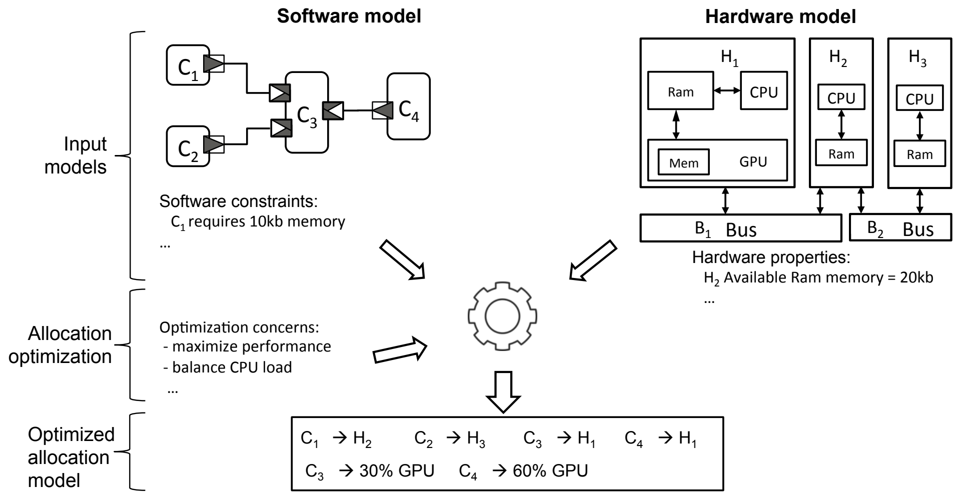


Fig. 1: Allocation optimization overview

II. Approach Overview

As illustrated in Fig. 1, the approach relies on three inputs to determine an optimized allocation scheme: a *software model*, a *hardware model* and a set of *optimization concerns*. The software model describes the system functionality as a set of interconnected components that are characterised by functional and extra-functional properties. The hardware model consists of computation nodes connected through communication buses. Computation nodes and communication buses are also annotated with functional and non-functional properties. Only two types of computation nodes are currently supported in the approach: simple nodes containing a memory block and a CPU (H_2 or H_3 in Fig. 1), and nodes containing a GPU and a dedicated GPU memory block in addition to the CPU and a memory block (H_1 in Fig. 1). Based on these inputs, the approach uses a mathematical solver to compute a possible solution.

In general, the software and hardware models that are used to describe complex heterogeneous embedded systems provide many details which are not important for the allocation itself. To abstract from this initial complexity, the input models are transformed into formal models. These models, as described below, only capture information directly relevant for the allocation and optimization process.

A. Software Model

The software model is specified as an undirected graph, where the vertices represent the software components and the edges represent the communication between the components. Software components and communication are annotated with functional and non-functional properties. An example of software component model is presented in Fig. 2, with the specification of the requirements for two of the components and a communication link.

A *Software Component* is described by the following properties:

- The amount of static *memory usage* the component needs. The property is expressed in kilobyte (kB).
- The *CPU usage* describes the workload usage of a component (e.g., information bits per clock cycle). Using a CPU workload unit reference (e.g., 1 unit = 32 bits of information each clock cycle), the property describes the component workload w.r.t. the CPU reference unit.
- The amount of global *GPU memory usage* the component needs. The property is expressed kilobyte (kB).
- The *GPU size* is defined as a sequence of alternative levels of GPU allocation, and is expressed in number of threads (T).
- The *performance* property is defined as a sequence of values, where each value represents the component performance for the corresponding value in the GPU size sequence.

Using different levels of GPU computation resources (e.g., threads) results in different performance values (e.g., execution time) [8]. However, the performance associated with different levels does not necessarily grow linearly. For example, a component with an execution time of 10 ms using 1000 threads does not necessarily finish in 5 ms if given 2000 threads. Various factors are included in the performance calculation (e.g., the automatic distribution of the software application over the GPU cores at runtime).

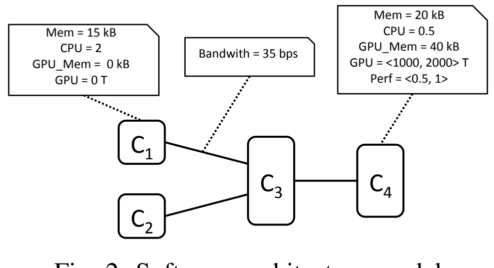


Fig. 2: Software architecture model

The *Communication Link* is the specification of interaction between two components. In our software model, we abstract the component connection information (e.g., ports, connectors and direction) using undirected edges between components. It is described by the *bandwidth* property which is the rate of data transfer between the two connected components. The property is expressed in bytes per second (bps).

B. *Hardware Model*

The heterogeneous platform is modelled as a bipartite graph as illustrated in Fig. 3. The graph has two distinct sets of vertices, where one set (the left hand side) represents the computation nodes, and the other set (the right hand side) represents the bus nodes. The edges have endpoints in different sets, and represents which computation node is connected to which bus node. In Fig. 3 two of the computation nodes and one of the bus node are annotated with examples of the properties that are used in the approach.

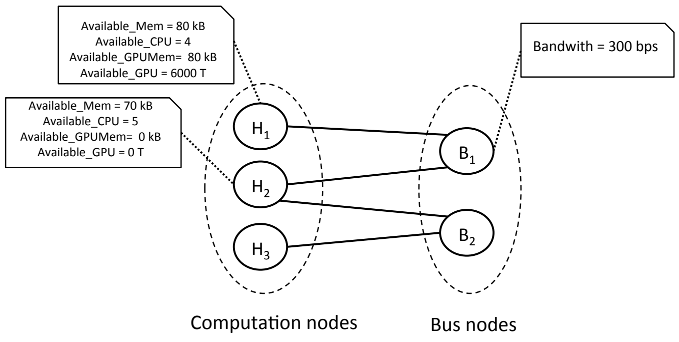


Fig. 3: Hardware architecture model

A *Hardware Computation Node* has the following properties:

- The static *memory size* available. The property is expressed in kilobytes (kB).
- The *CPU capacity* describes the workload of a node (e.g., information bits processed in one clock cycle). Using a CPU reference workload unit (e.g., 1 unit = 32 bits of information each clock cycle), the property describes the CPU workload w.r.t. to the reference unit.
- The global *GPU memory size* available. The property is expressed in kilobytes (kB) and represents the size of the GPU global memory.
- The *GPU size*. The property is expressed in number of threads (T).

The *Hardware Bus Node* is the representation of the communication channel (e.g., a CAN bus). It is described by the *bandwidth* property which is the maximum rate at which data can be transmitted between two connected nodes. The property is expressed in bits per second (bps).

C. *Optimization Concerns*

Determining feasible allocation schemes requires knowledge on the functional and non-functional properties of the system, the hardware architecture and the system requirements. However, not all feasible allocation schemes are equivalent. Allocation being a cross-cutting concern, applying a certain allocation scheme can positively (or negatively) influence the non-functional properties of the system. It is thus necessary to establish which of the feasible solutions are the most suitable, i.e. one must decide which criteria should be optimized and how.

The optimization process deals with the performance optimization (maximize the GPU distribution) and the fitness of the hardware resource utilization (such as the utilization load of each CPU node, the memory load or the bus communication load). The performance optimization aims at maximizing the performance of the allocation model based on a developer ranking the importance of the software components. For example, a components that needs to process huge amounts of data may have a lower performance if it only has access to a small part of the GPU. Conversely, it may have a higher performance if it uses more of the GPU. The optimization function maximizes the distribution of the GPU among components according to their importance ranking.

There are different concerns which can be used for the fitness of the hardware resource utilization such as "maximize the CPU usage of a given node" or "minimize the number of hardware nodes used in the system". In this paper we focus on the balancing of hardware resources as the fitness concern.

D. *Allocation Scheme*

The allocation scheme contains the result of the optimization process, i.e., a feasible and optimized mapping of each software component onto a hardware computation node that satisfies the constraints of the system and its requirements. In addition to the component-node mapping, the scheme describes how the GPU workload is distributed among the components. This work being focused on the formalization of the optimization allocation, our current allocation model is straightforward: software components and computation nodes are represented by a unique name and the mapping is seen as a placement of components onto nodes. A complete allocation model would include, however, additional information such as the specification of detailed memory mapping, allocation of a component to a given processor and, if is the case, to the associated GPU within a node.

III. Allocation Optimization Model

The formal model which captures the system characteristics and constraints, and describes the definition of the allocation optimization challenge, is defined as follows:

A. *Input*

1) A set C of n software components, and seven functions *link* : $W \to \mathbb{R}$, where W contains unordered pairs from C, *memR* : $C \to \mathbb{R}$, *cpuR* : $C \to \mathbb{R}$, *memgpuR* : $C \to \mathbb{R}$, *gpuR* : $C \to \mathbb{N}^{\mathbb{N}}$ and *perf* : $C \to \mathbb{N}^{\mathbb{R}}$, *perfImp* : $C \to \mathbb{R}$, where:

$$\mathit{link}(c_i, c_j) = \begin{cases} 0, \textit{ when } c_i = c_j \\ 0, \textit{ when } c_i \textit{ not connected to } c_j \\ \textit{bitrate between } c_i \textit{ and } c_j\textit{, when} \\ \quad c_i \neq c_j \textit{ are connected} \end{cases}$$

$\mathit{memR}(c)$	$=$	*memory required by c*
$\mathit{cpuR}(c)$	$=$	*CPU workload required by c*
$\mathit{memgpuR}(c)$	$=$	*GPU memory required by c*
$\mathit{gpuR}(c)$	$=$	*a sequence* $\langle k_1, \ldots, k_i \rangle$ *of alternative GPU levels*
$\mathit{perf}(c)$	$=$	*a sequence* $\langle p_1, \ldots, p_i \rangle$ *of performance values, where* $p_x \in [0,1]$ *for* $1 \leq x \leq i$
$\mathit{perfImp}(c)$	$=$	*the importance of component c, where* $\mathit{perfImp}(c) \in [0,1]$

2) A set H of k hardware computation nodes, a set B of m bus nodes, five functions $\mathit{memA} : H \to \mathbb{R}$, $\mathit{cpuA} : H \to \mathbb{R}$, $\mathit{memgpuA} : H \to \mathbb{R}$, $\mathit{gpuA} : H \to \mathbb{R}$, $\mathit{brtA} : B \to \mathbb{R}$, and a relation $\mathit{node\_link} \subseteq H \times B$, where:

$\mathit{memA}(h)$	$=$	*available memory on node h*
$\mathit{cpuA}(h)$	$=$	*available CPU capacity on node h*
$\mathit{memgpuA}(h)$	$=$	*available GPU memory on node h*
$\mathit{gpuA}(h)$	$=$	*available GPU size on node h*
$\mathit{brtA}(b)$	$=$	*available bitrate on bus b*
$\mathit{node\_link}(h, b)$	$=$	*the connection between the node h and bus b*

3) The weight factors of the fitness function: $w_{mem}, w_{cpu}, w_{bitrate}, w_{perf} \in [0,1]$. These are described in more details in Section III-C.

B. Constraints

Given this input, the goal of the allocation optimization is to find the functions $\mathit{alloc} : C \to H$ (i.e., mapping the components to nodes) and $\mathit{gpualloc} : C \to \mathbb{N}$ (i.e., distributing the GPU resources among components), such that the following constraints are satisfied:

1) The summed required memory of components placed on the same node should not exceed the available node memory.

$$\forall h \in H\,(\mathit{memAll}(h) \leq \mathit{memA}(h))\text{, where}$$
$$\mathit{memAll}(h) = \textstyle\sum_{c \in \{c | c \in C \wedge \mathit{alloc}(c) = h\}} \mathit{memR}(c)$$

2) The summed required CPU of components placed on the same node should not exceed the available node CPU.

$$\forall h \in H\,(\mathit{cpuAll}(h) \leq \mathit{cpuA}(h))\text{, where}$$
$$\mathit{cpuAll}(h) = \textstyle\sum_{c \in \{c | c \in C \wedge \mathit{alloc}(c) = h\}} \mathit{cpuR}(c)$$

3) The summed required GPU memory of components allocated to the same GPU unit should not exceed the available memory.

$$\forall h \in H\,(\mathit{gmem}(h) \leq \mathit{memgpuA}(h))\text{, where}$$
$$\mathit{gmem}(h) = \textstyle\sum_{c \in \{c | c \in C \wedge \mathit{alloc}(c) = h \wedge \mathit{gpualloc}(c) > 0\}} \mathit{memgpuR}(c)$$

4) The GPU size allocated to a component should be one of the alternatives for that component .

$$\forall c \in C\,(\mathit{gpualloc}(c) \in \mathit{gpuR}(c))$$

5) The sum of the required GPU size of components placed on the same node should not exceed the available GPU size of that node.

$$\forall h \in H\,(\mathit{gpuAll}(h) \leq \mathit{gpuA}(h))\text{, where}$$
$$\mathit{gpuAll}(h) = \textstyle\sum_{c \in \{c | c \in C \wedge \mathit{alloc}(c) = h\}} \mathit{gpualloc}(c)$$

6) The sum of the required bitrate of components which are placed on different nodes but connected to the same bus node, should not exceed the available bus node bitrate.

$$\forall b \in B\,(\mathit{busAll}(b) \leq \mathit{brtA}(b))\text{, where}$$
$$\mathit{busAll}(b) = \textstyle\sum_{\{c_i, c_j\} \in \mathit{cnct}(b)} \mathit{link}(c_i, c_j) \text{ for}$$
$$\mathit{cnct}(b) = \{\{c_i, c_j\} | c_i, c_j \in C \wedge \mathit{node\_link}(\mathit{alloc}(c_i), b) \wedge$$
$$\mathit{node\_link}(\mathit{alloc}(c_j), b) \wedge \mathit{alloc}(c_i) \neq \mathit{alloc}(c_j)\}$$

C. Optimization functions

Our optimization process considers four aspects: memory balancing, CPU balancing, bitrate balancing and GPU performance. Each aspect is represented by a fitness function and a weight factor defined by the developer.

$$\mathit{fitness}(\mathit{alloc}, \mathit{gpualloc}) = w_{mem} * F_{mem}(\mathit{alloc}) + w_{cpu} * F_{cpu}(\mathit{alloc}) + w_{bus} * F_{bus}(\mathit{alloc}) + w_{perf} * F_{perf}(\mathit{gpualloc})$$

$$w_{mem} + w_{cpu} + w_{bus} + w_{perf} = 1$$

The weight factors describe which concern has a higher importance for the application or is used to exclude one or several functions from the optimization process. For example, if we want to optimize the GPU performance and balance the CPU load of the system, we set w_{mem} and $w_{bitrate}$ to 0, and define the rest of the weights values according to their importance.

In the following description, we consider the case for balancing the hardware resources (memory, CPU and communication) and maximize the GPU distribution.

1) CPU balancing

The CPU fitness function balances the workload of the entire hardware system as follows. First, the CPU usage of each node is derived using formula (1), by dividing the sum of all components CPU load placed on the same node with the node available workload. Then, using formula (2), we compute the system CPU workload. To balance the CPU load for a single node, we calculate the absolute value of the difference between the average system usage and the node usage. Applying the same principle to a system with k nodes, the balanced CPU workload is given by formula (3).

$$\mathit{useCpu}(h) = \frac{\mathit{cpuAll}(h)}{\mathit{cpuA}(h)} \tag{1}$$

$$\overline{\mathit{useCpu}} = \frac{\sum_{h=1}^{k} \mathit{useCpu}(h)}{k} \tag{2}$$

$$F_{cpu}(alloc) = 1 - \frac{\sum_{h=1}^{k} \left|\overline{useCpu} - useCpu(h)\right|}{k} \quad (3)$$

2) Memory balancing

The memory fitness function balances the memory usage of the system, in the same manner as for the CPU system workload. First, we calculate the memory usage of each node, using the formula (4). The system memory average usage is obtained by formula (5). The final memory balancing function is described in formula (6).

$$useMem(h) = \frac{memAll(h)}{memA(h)} \quad (4)$$

$$\overline{useMem} = \frac{\sum_{h=1}^{k} useMem(h)}{k} \quad (5)$$

$$F_{mem}(alloc) = 1 - \frac{\sum_{h=1}^{k} \left|\overline{useMem} - useMem(h)\right|}{k} \quad (6)$$

3) Communication balancing

To balance the communication of a system, we start by computing the communication usage of each bus node, using formula (7). The system average of the communication usage is obtained by formula (8). Formula (9) is balancing the system communication usage.

$$useBus(b) = \frac{bussAll(b)}{brtA(b)} \quad (7)$$

$$\overline{useBus} = \frac{\sum_{b=1}^{m} useBus(b)}{m} \quad (8)$$

$$F_{bus}(alloc) = 1 - \frac{\sum_{b=1}^{m} \left|\overline{useBus} - useBus(b)\right|}{m} \quad (9)$$

4) Performance optimization

The performance optimization function distributes the GPU workload to components in such a way that a component with a higher importance will access more computation resources than a component with a lower importance. The importance weights, previously specified by the developer in the input section, are defined only for components which require GPU. Through the importance weights, the developer classifies the component access to GPU by their relevance. By maximizing the performance function described in formula (10), higher importance components are given more access to the GPU.

$$F_{perf}(gpualloc) = \frac{\sum_{c \in C} perfImp(c) * p_j}{n} \quad (10)$$

where $perf(c) = \langle p_1, \ldots, p_i \rangle$ and *gpualloc(c)* is the j^{th} element in *gpuR(c)*

IV. Translation to Solver

To compute solutions for our optimization model, we introduce SCIP [1] which is a mixed-integer programming (MIP) solver and a framework for constraint integer programming. Our optimization model can be seen as a mixed-integer nonlinear programming model, where we can minimize or maximize several functions (e.g., CPU load on a node, system memory usage) subject to a finite number of constraints of integer variables. Translating the model into a solver is done using a standard format called MPS (Mathematical Programming System), when almost all available MPS solvers can interpret today. ZIMPL [9], used as an intermediary language, mitigates the conversion of the model to an MPS format.

In the input part of the ZMPL mode, the hardware and software models are constructed, their components and nodes initialized, and the communication links, performances and importance weights defined. Two boolean array variables are defined to hold the result of the allocation, one for mapping the components to host and the other for the GPU distribution over the components, as follows.

var allocate[CH] boolean;
var distribute[CT] boolean;

The *allocate* array contains boolean variables of all possible mapping combinations between components and hosts (using a Cartesian product set CH of components and hosts). The *distribute* array is constructed similar. It contains boolean variables of all possible combinations between components and alternatives GPU levels (using a Cartesian product set CT of components and their alternative GPU levels). In the end, the two arrays will describe the solution by displaying values of 1 for the feasible component-node and component-GPU alternative level mapping solutions.

The constraints translation follows the mathematical model we defined in Section III. For example, the following instruction presents the ZIMPL constraint over the sum of component memory usage.

```
forall < h > in H do
    (sum < c > in C : mem_comp[c] * allocate[c,h]) <=
      mem_node[h];
```

For each node h, we condition the sum of components memory usage $mem\_comp[c]$ placed on the same node (enforced by the boolean value $allocate[c,h]$) to be less or equal to the available node memory $mem\_node[h]$. All other constraints (CPU workload, bandwidth, etc) are translated in a similar form.

In the last part of the model, we translate the fitness function where we balance the resource usage and optimize the GPU distribution. The following instruction describes a small part of the fitness function translation, representing the GPU distribution to components based on their importance.

```
Fperf = sum < c,t > in CT:
    importance[c] * perf[c, t] * distribute[c, t];
```

Once the ZIMPL translation is finished, the solver receives it as an input, and computes the *allocate* and *distribute* arrays.

V. Evaluation

The evaluation consists of two parts. The first part illustrates the applicability of the approach on a concrete example of an academic embedded system, and the second the scalability of the approach using different optimization criteria.

A. *Application to an autonomous underwater robot*

To examine the practical usage of our optimization model, we use an underwater robot with stereo vision as an example of a complex heterogeneous CPU-GPU embedded system. The robot is developed at Mälardalen University, Sweden, as demonstrator for the RALF3 research project [2]. The purpose of the robot is, based on the vision system, to autonomously operate under water in searching and tracking various objects. The hardware and software models described below are based on the original hardware and software models of the robots. They have, however, been adjusted to allow evaluating the work presented in this paper.

The robot's hardware platform is composed of three boards connected by a CAN bus. In addition, the robot also contains various sensors and actuators, such as cameras, pressure and ultrasonic sensors and motors. Fig. 4a presents the corresponding hardware model, in which H_3 is a simple CPU computation node, H_1 and H_2 are two identical complex computation nodes (i.e. CPU-GPU nodes). The model elements have been annoted with corresponding extra-functional properties.

The functionality of the robot is originally modelled as a composition of software components connected through interfaces. For the purpose of the work, this original model has been flatten down and transformed to the formal representation presented in Section II.A. The result of this transformation is depicted in Fig. 4b. In this model, the Decision Center is the main component: it gets and sets the configuration data (e.g., the color calibration parameters, the water pressure), and decide the execution order of the different missions according to the data captured by the sensors. The Align component is responsible for aligning the robot with a given object, an underwater path for example. Based on the results of the analysis of the images performed by the Vision Manager component, the Align component must first calculate the parameters to move the robot in the correct direction and then effectively move the robot by calling the Movement Navigation component which provides the movement commands for the robot (e.g., move left, move right, etc). As shown in the extra-functional property annotations, the Align component does not require any massive parallel GPU computation but has a high CPU workload and memory usage. On the other hand, the Object Detector component, controlled by the Vision Manager component, requires high GPU computation resources for processing the camera images. Three GPU alternative levels and their corresponding performances are also provided for the Object Detector.

In addition to the model constraints (see Section III-B), several component mapping restrictions must be introduced. From the hardware model, only the CPU-GPU boards are connected to a vision camera. This implies that the software component that controls a vision camera can only be placed on a board containing the corresponding camera. Similarly, a peripheral component that implements the controls for a given sensor (resp. actuator) should be placed on a board that has the device. In other words, Peripherical1 which controls the IMU should be placed on node H_1 and the Movement Navigation in charge of the motors should be allocated to H_3.

In the following example, we use only the optimization performance concern that is the distribution of GPU among the vision components. There are four components which require GPU: Vision Manager, Object Detector, Front Filter and Bottom Filter. Being the component which manages the vision process, Vision Manager receives the highest importance, followed by the Object Detector importance.

TABLE I: Optimized allocation scheme

Component	Node	GPU
Align	H_3	
Bottom Camera	H_2	
Bottom Filter	H_2	3000 T
Data Recorder	H_3	
Decision Center	H_3	
Front Camera	H_1	
Front Filter	H_1	2000 T
Interaction Center	H_3	
Movement Navigation	H_3	
Object Detector	H_1	4000 T
Peripheral1	H_1	
Peripheral2	H_2	
Peripheral3	H_3	
Vision Manager	H_2	2000 T

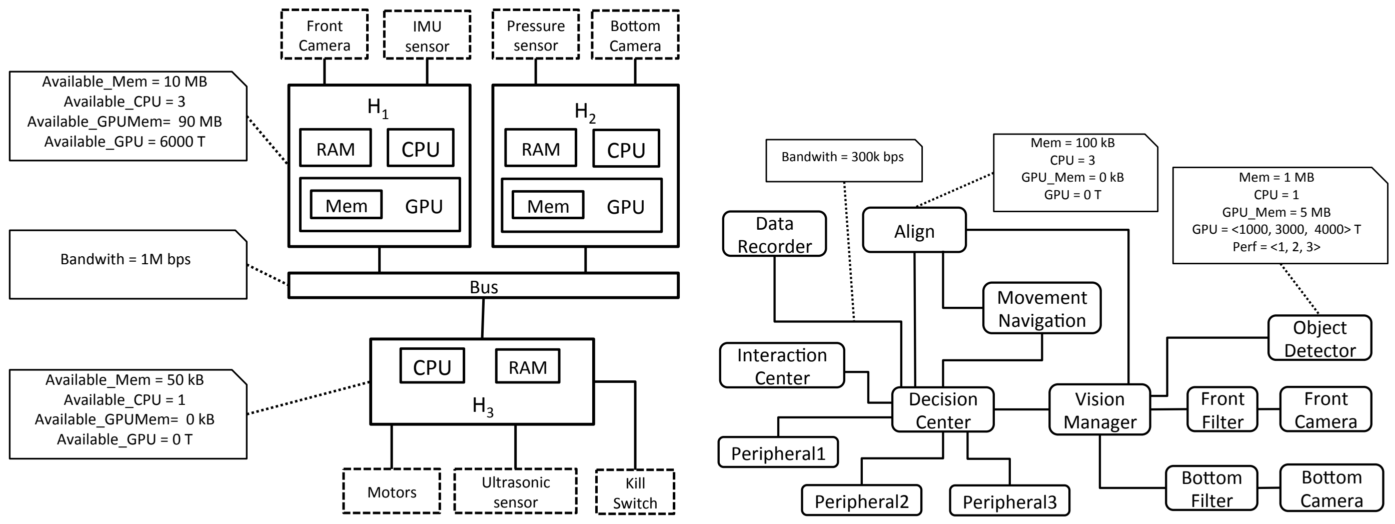
Table I presents the solution found by the solver. The Object Detector component, having a higher importance than the Front Filter component, will receive the maximum level of its GPU request. On the other complex node H_2, the Vision Manager component, having a high importance, receives the maximum GPU level request while the Bottom Filter component, having the rest of the GPU resources at its disposal, receives the highest level of its GPU request. For this simple example consisting of 14 software components and three hardware nodes, the solution was found in few milliseconds. The platform on which the solver was executed is an Optiplex 780 desktop with an Intel Core 2 Duo processor and 2 GB of memory.

TABLE II: Optimization time

	Components (GPU comp)	Time (seconds)
$F_{balance}$	30 (10)	1.23
	35 (12)	1.50
	40 (14)	2.65
	45 (16)	12.03
F_{perf}	30 (10)	55
	35 (12)	171
	40 (14)	484
	45 (16)	12046

B. *Scalability*

A set of experiments were conducted in order to evaluate the growth of the optimization time. We implemented a generator for random software and hardware input models (i.e., sets of components and nodes with their properties) of different sizes. Using the generator, sets of four cases are computed. The cases have from 30 to 45 software components, a random number of connections, and the same hardware model: 7 nodes from which 3 are complex, and 1 bus node. Table II



(a) Hardware model (b) Simplified software model

Fig. 4: Underwater robot demonstrator

presents the solver optimization time using our generated sets while running two different optimization concerns. The first optimization concern $F_{balance}$ balances the hardware resources (criterion 1, 2 and 3 from Section III-C), while the second F_{perf} computes schemes based on the GPU optimization performance criterion. For each case, an average of 10 simulations is computed. Executing on the same hardware platform used in the previous validation example, the solver is sufficiently fast in computing solutions for systems with up to 40 software components. There is a large change in solving time while jumping to systems with 45 components.

VI. Related Work

A large body-of-knowledge exists on software optimization, with works such as [4] and [3] which contribute to summarize it. Yet, only few works consider other computation units than CPUs and even fewer, heterogeneous computation nodes that use combination of CPUs and GPUs.

Several works present task assignment onto CPU-based computing systems. Among them, in [13], the authors present a deployment optimization method for automotive industry. The allocation strategy is intended only for vehicle hardware platforms composed of ECUs with different memory capacity and processing power. The software is allocated onto the hardware platform considering the data transmission reliability and communication overhead attributes. This work focuses only on automotive computation ECU nodes, considering properties specific to this domain (i.e., ECU capacity, speed, failure rate or data transmission reliability). Ucar et al. propose a method [17] for task allocation onto processors with different powers, in order to minimize the system utilization. In [11], the authors introduce a task allocation model for distributed systems, with the goal of balancing the utilization of each processor. The authors of [18] generate, using a solver, optimized assignments of tasks to CPU cores taking in consideration the local memory constraints and criticality constraints of tasks. This work focuses on safety-critical aspects and multi-core CPU system allocation. Although we are abstracting the software application as a component-based model, our fitness function uses similar principles as presented in the previous papers for minimizing the system utilization. In [19], the authors design and evaluate load-sharing policies for CPU and memory in heterogeneous distributed systems. The interesting part for us is how they formalize the CPU/memory weights for heterogeneity. We are using a similar formalization but we extended our work to cover also the communication property.

A similar optimization problem is addressed by Svogor et al. by using a genetic algorithm based method [16]. This work discusses a possible approach to compute allocation schemes for hardware platforms with CPUs, GPUs and FPGAs nodes.

An analysis on the execution time of massively parallel GPU programs is presented by Hong et al. [8]. The authors explain how, among others factors, a system can have a better execution time by using a high number of threads (grouped in warps). The component performance from our optimization model is constructed using the same principle, being proportional with the number of threads used by the component.

The AQOSA toolkit [10] describes an automated optimization process which, based on some initial input software architectures, generates alternative architecture models. In the optimization process, various metrics are considered such as processor utilization, data flow latency, etc. These models are analyzed and evaluated, helping the software architects in reducing the work for modeling. Although the AQOSA tool is addressing the optimization problem, it uses a different approach (MDE) than ours for improving different quality attributes of the system.

VII. Conclusion and Future Work

In this paper, we have presented our initial work to optimize allocation of software applications onto CPU-GPU embedded hardware architectures. The work has been focused on identifying key allocation parameters and specifying a suitable mathematical model to the allocation problem. Relying on the software and hardware properties, several fitness functions for balancing the resources and a criterion to maximize the distribution of the GPU resources, a solver searches for optimized allocation schemes.

To evaluate the practical usage of the optimization model on a real example, an underwater robot is used as a demonstrator. The allocation scheme computed by the solver, presents the component-node mapping and the distribution of GPU computation. Also, as part of the evaluation, a set of experiments were conducted to present the solver running time while computing allocation schemes. Solutions for medium-size problems (about 45 software components and 7 computation hosts) were calculated in approximate 3.5 hours on standard hardware. The exponential increasing of time relative to the expansion of the problem complexity represents the limitation of using a solver in finding feasible allocation schemes.

Being in its inception phase, the approach needs to be further developed. We first intent to alleviate some of the assumptions and constraints we set for the work in adding new types of computation nodes (e.g. FPGA), introducing new optimization criteria (scenario-based optimization) and supporting additional non-functional properties (e.g. dynamic memory usage, throughput and response time). Also, enabling deployment within a node would bring more flexibility to our model in allowing, for example, a component to use more CPU usage to compensate the lack of GPU. Covering a more detailed GPU characteristics (e.g., local registers, shared memory) in a future optimization model will result in a more precise allocation scheme. As part of a future evaluation target, various experiments will cover solver running time on different combinations of fitness function concepts (e.g., balancing CPU load while optimizing GPU performance distribution). We also envisage to integrate the approach with some well-known modelling languages such as MARTE to provide a more complete model-driven engineering (MDE) approach which will ease the workload of software architects.

Acknowledgment

Our research is supported by the RALF3 project (http://www.mrtc.mdh.se/projects/ralf3/) through Swedish Foundation for Strategic Research (SSF).

References

[1] T. Achterberg. *Constraint Integer Programming*. PhD thesis, TU Berlin, Germany, 2007.

[2] C. Ahlberg, L. Asplund, G. Campeanu, F. Ciccozzi, F. Ekstrand, M. Ekström, J. Feljan, A. Gustavsson, S. Sentilles, I. Svogor, and E. Segerblad. The black pearl: An autonomous underwater vehicle. Technical report, Mälardalen University, June 2013. Published as part of the AUVSI Foundation and ONR's 16th International RoboSub Competition, San Diego, CA.

[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, 2013.

[4] S. Balsamo, A. D. Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: A survey. *IEEE Transactions on Software Engineering*, 30(5):295–310, 2004.

[5] S. K. Baruah. Task partitioning upon heterogeneous multiprocessor platforms. In *Real-Time and Embedded Technology and Applications Symposium, 2004. Proceedings. RTAS 2004. 10th IEEE*, pages 536–543, 2004.

[6] S. Che, J. Li, J. W. Sheaffer, K. Skadron, and J. Lach. Accelerating compute-intensive applications with gpus and fpgas. In *2008 IEEE Symposium on Application Specific Processors (SASP)*. IEEE, 2008.

[7] D. Geronimo, A. M. Lopez, A. D. Sappa, and T. Graf. Survey of pedestrian detection for advanced driver assistance systems. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 32(7):1239–1258, 2010.

[8] S. Hong and H. Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. *SIGARCH Comput. Archit. News*, 37(3):152–163, June 2009.

[9] T. Koch. *Rapid Mathematical Programming*. PhD thesis, Technische Universität Berlin, 2004.

[10] R. Li, R. Etemaadi, M. T. M. Emmerich, and M. R. V. Chaudron. An evolutionary multiobjective optimization approach to component-based software architecture design. In *IEEE Congress on Evolutionary Computation*, pages 432–439. IEEE, 2011.

[11] P.-Y. R. Ma, E. Lee, and M. Tsuchiya. A task allocation model for distributed computing systems. *IEEE Transactions on Computers*, 31(1):41–47, 1982.

[12] P. Michel, J. Chestnutt, S. Kagami, K. Nishiwaki, J. Kuffner, and T. Kanade. Gpu-accelerated real-time 3d tracking for humanoid locomotion and stair climbing. In *Intelligent Robots and Systems, 2007. IROS 2007. IEEE/RSJ International Conference on*, pages 463–469. IEEE, 2007.

[13] I. Moser and S. Mostaghim. The automotive deployment problem: A practical application for constrained multiobjective evolutionary optimisation. In *IEEE Congress on Evolutionary Computation*, pages 1–8. IEEE, 2010.

[14] G. L. Nemhauser and L. A. Wolsey. *Integer and combinatorial optimization*, volume 18. Wiley New York, 1988.

[15] H. Nguyen. *Gpu Gems 3*. Addison-Wesley Professional, first edition, 2007.

[16] I. Svogor, I. Crnkovic, and N. Vrkic. Multi-criteria software component allocation on a heterogeneous platform. In *In: Proc. of 35th International Conference on Information Technology Interfaces*. IEEE Computer Society Press, 2013.

[17] B. Ucar, C. Aykanat, K. Kaya, and M. Ikinci. Task assignment in heterogeneous computing systems. *J. Parallel Distrib. Comput.*, 66(1):32–46, Jan. 2006.

[18] S. Voss and B. Schtz. Deployment and scheduling synthesis for mixed-critical shared-memory applications. In J. W. Rozenblit, editor, *ECBS*, pages 100–109. IEEE, 2013.

[19] L. Xiao, X. Zhang, and Y. Qu. Effective load sharing on heterogeneous networks of workstations. In *In: Proc. of International Symposium on Parallel and Distributed Processing*, pages 431–438. IEEE Computer Society Press, 2000.

[20] F. Xu and K. Mueller. Real-time 3d computed tomographic reconstruction using commodity graphics hardware. *Physics in medicine and biology*, 52(12):3405, 2007.