Scavenging Run-time Resources to Boost Utilization in Component-based Embedded Systems with GPUs

Gabriel Campeanu, Saad Mubeen
Mälardalen Real-Time Research Center (MRTC),
Mälardalen University, Västerås, Sweden
Email: {gabriel.campeanu, saad.mubeen}@mdh.se

***Abstract*—Many modern embedded systems with GPUs are required to process huge amount of data that is sensed from their environment. However, due to some inherent properties of these systems such as limited energy, computation and storage resources, it is important that the resources should be used in an efficient way. For example, camera sensors of a robot may provide low-resolution frames for positioning itself in an open environment and high-resolution frames to analyze detected objects. In this paper, we introduce a method that, when possible, scavenges the unused resources (i.e., memory and number of GPU computation threads) from the critical functionality and distributes them to the non-critical functionality. As a result, the overall system performance is improved without compromising the critical functionality. The method uses a monitoring solution that checks the utilization of the system resources and triggers their distribution to the non-critical functionality whenever possible. As a proof of concept, we realize the proposed method in a state-of-the-practice component model for embedded systems. As an evaluation, we use an underwater robot case study to evaluate the feasibility of the proposed solution.**

Keywords–GPU; embedded system; component-based software development; CBD; model-based development; MBD; resource utilization; monitor.

I. INTRODUCTION

This paper substantially extends the authors' previous work [1]. Embedded systems are found in almost all contemporary electronic products. Their applications range from simple and small-sized products, e.g. a wireless controlled toy car or an airplane to very complex and large-sized systems such as premium cars and airplanes. Many modern embedded systems are developed to process huge amount of data that is originated from the interaction with their environments. For example, the Google autonomous car processes around 750 MB of data per second [2]. Massive computing power and parallel execution of software is required to process such a large amount of data. The traditional embedded systems are unable to handle these data-intensive applications, mainly due to reduced computational power and support for parallel execution of software.

Graphics Processing Units (GPUs) offer a promising solution to deal with this challenge. A GPU supports a parallel execution model, which allows multiple data to be processed in parallel. However, a GPU cannot be used in isolation, i.e., the GPU needs a Central Processing Unit (CPU) to activate the activities (threads) that are intended to run in parallel on it. Thanks to the recent advances in the semiconductor and chip industry, there are several vendors that manufacture heterogeneous computing platforms which contain a combination of CPUs, GPUs and other computation resources on the same board. For example, vendors such as NVIDIA, AMD and Samsung provide their own embedded heterogeneous solutions on the same board such as NVIDIA Jetson TK1 [3], AMD R-464L [4] and Samsung Exynos 8 [5] respectively.

The amount of data captured by an embedded system from its environment can significantly impact the management of its resources (e.g., memory and computation power), which in turn can impact its performance. One way to optimize the resource usage is to collect variable stream-size of data from the sensors depending upon different situations. For example, the ProcImage500-Eagle camera sensor [6] can be configured to capture low- or high-resolution frames depending upon the environment. For example, a robot fitted with such a camera may use low-resolution data frames to examine its position in an open environment. On the other hand, the robot may use high-resolution frames to inspect the target objects in a detailed manner. The high-resolution frames require larger memory footprints and more computation power (and energy) to be processed by the GPU. Whereas, the low-resolution frames are delivered with faster frame rate, occupy less memory and require lower computation power for GPU processing.

The system resources in many embedded systems are shared between non-critical and critical functions. The non-critical functions are not constrained by any resource requirements. Hence, these functions are expected to provide the best-effort service. Whereas, the critical functions are constrained by resource requirements, often stringent, that must be met during the execution of the system. Hence, it is ensured that all the system resources that are required by the critical functions are always available to them. For example, a vision system of a robot represents a critical function. This system is designed in such a way that it is always guaranteed enough resources to process the high-resolution frames. Even when the cameras provide low-resolution frames, the system still occupies the same amount of resources as if it were processing the high-resolution frames. As a consequence, the system resources are wasted when they are not needed by the critical function. We argue that the non-critical functions can benefit from the

resources of the critical functions during the intervals when they are not used. For example, when the robot utilizes low-resolution frames, a non-critical function such as a logger system can benefit from the extra memory which is not being used by the vision system.

This paper provides a method to automatically compute the unused resources in the critical part of the system. The method then distributes the computed resources to the non-critical parts of the system. The proposed method is based on a run-time monitoring engine that monitors the critical part of the system to detect any changes in its resource requirements. If the engine detects a reduction in the resource requirements of the critical system, it triggers the proposed method to calculate the unused memory based on the actual resource usage by the critical part of the system. The information regarding the amount of available memory is provided to non-critical part of the system, which can benefit from the available extra resource.

The initial conference paper [1] presented a method to improve the resource utilization in embedded systems based on the system memory as the only run-time resource. The submitted journal paper extends our method by considering other run-time resources such as the GPU computation threads together with the system memory to further boost the resource utilization of embedded systems with GPUs.

The rest of the paper is organized as follows. Section II describes the background context. Section III formulates the problem and describes it with the help of a case study. The overview of our solution is described in Section IV-A and its realization is presented in Section IV-B. Section IV-C discusses the implementation of the solution. The evaluation of our method applied to the case study is discussed in Section V. Finally, Section VI introduces the existing work related to our approach, while Section VII concludes the paper.

II. Background

The section provides background information on GPUs and some software development strategies for embedded systems.

A. GPUs

GPUs were developed in 90s and were employed only in graphic-based applications. With the passage of time GPUs became more popular due to increase in their computation power and ease of use. As a result, GPUs have been utilized in different types of applications, even becoming the general-purpose processing units referred to as GPGPUs [7]. For example, cryptography applications [8] and Monte Carlo simulations [9] use GPU-based solutions. Equipped with a parallel architecture, the GPU may employ thousands of computation threads at a time through its multiple cores. Compared to the traditional CPU, the GPU delivers an improved performance with respect to processing multiple data in parallel. For example, simulation of bio-molecular systems have achieved 20 times speed-up on GPU [10].

One of the GPU characteristics is that it cannot function without the help of a CPU. The CPU is considered as the brain of the system that triggers all the activities related to GPU, such as the execution of functionality onto GPU. The GPU often has its own private memory system, which requires data to be copied from one memory system to the other in order to be accessed by the corresponding processing unit. Note that even if the CPU and GPU are integrated on the same physical chip, the chip's memory is divided in two distinct memory spaces, one for each processing unit. Figure 1 presents an example of an embedded platform that has the CPU and GPU integrated on the same chip. While the CPU is equipped with two cores (i.e., two execution threads), the GPU has several cores characterized by hundred of execution threads. The physical memory is divided in two sections, each processing unit accessing its private memory section. Due to the reduced physical size, lower energy usage and costs, this embedded solution is one of the commonly used solution in the industry. These reasons motivate us to consider the platform that has distinct memory spaces for CPU and GPU.

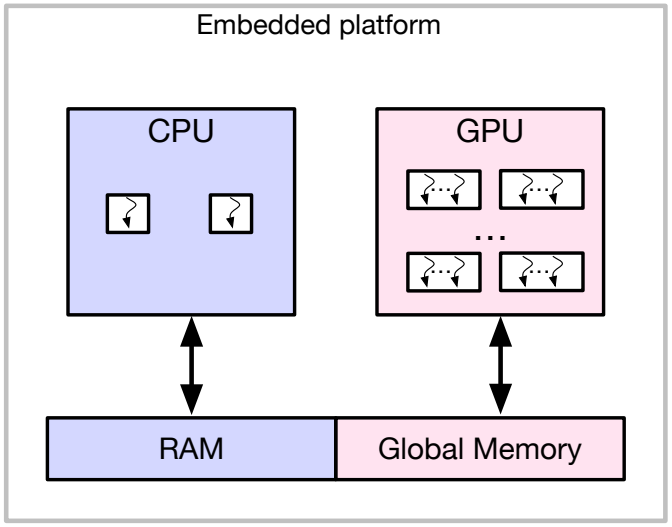


Figure 1. A CPU-GPU embedded platform.

B. Model- and Component-based Software Development

The software complexity in embedded systems has significantly increased in various domains, e.g., the automotive and robotic domains. The software development techniques that are based on the principles of model-based engineering (MBE) and component-based software engineering (CBSE) have proven efficient in dealing with the software complexity [11], [12]. CBSE is also successfully employed to develop robotic applications [13]. Using these paradigms, models are used throughout the development process. These software models allow the development of applications by connecting software units, called *software components*. CBSE promotes the (re-)use of the same component in different contexts. One benefit of adopting CBSE is the development efficiency that is achieved through reusable software components. A key concept of CBD is the

encapsulation, where all the information of a component is encapsulated inside, hidden from anything outside. A way to access the encapsulated information is through *interfaces*. In this work, we focus on *port-based interfaces*, where the ports are access points of software components. MBE and CBSE have been successfully adopted by the industry through various component models.

When a software component is developed, the specifications of a component model are followed. For example, Component Object Model (COM) specifies that all of the COM components should be constructed with an *IUnknown* interface [14]. The component model also describes the way its components interact and how they are combined in systems. There exists many component models, some designed for particular domains (e.g., automotive) and other built on specific technological platforms (e.g., Enterprise Java Beans [15]). Note that component models follow various interaction styles that are suitable for different types of applications [16]. We mention the *request-response* and *sender-receiver* interaction styles that are utilized in AUTOSAR [17] component model when developing automotive applications. Another style utilized by e.g., Rubus Component Model (RCM) [18] and IEC 61131 [19] component models, is the *pipe-and-filter* interaction style. This particular style is characteristic to streaming-of-event type of applications and allows an easy mapping between the flow of system actions and control specifications, characteristic to real-time and safety-critical applications.

The existing component models that can be used to build stream-of-event applications, e.g., RCM, AUTOSAR, IEC 61131 and ProCom[20], face the challenge of dealing with (streaming) data that can change its memory footprint on-the-fly. For example, RCM defines that its components use the same fixed memory footprint throughout the execution of the application. In order to ensure the required resources to the critical functionality, resources are assigned to each RCM software component, with respect to its worst-case resource demand for the entire system execution. Therefore, RCM and similar component models (discussed above) do not support any mechanism to release the resources when they are not required (by the critical part of the system).

In this paper we consider RCM as a proof of concept for the implementation of our solution. A Rubus software component, also known as the *software circuit*, is the lowest-level hierarchical element. It is characterized by input ports and output ports. A feature of RCM is that there is a clear separation between the control flow and data flow. Therefore, a software circuit has two types of ports, i.e., data and trigger ports. Figure 2 presents three connected Rubus components, each equipped with one single (input and output) trigger port, and one or several (input and output) data ports. The software circuit uses the read-execute-write execution semantics. Initially in an idle state, a component is activated through its input trigger port. It starts by reading the data from its input data ports, then it executes its functionality, followed by writing the results to the output data ports. Finally, it transfers the control through its output trigger port and re-enters the idle state.

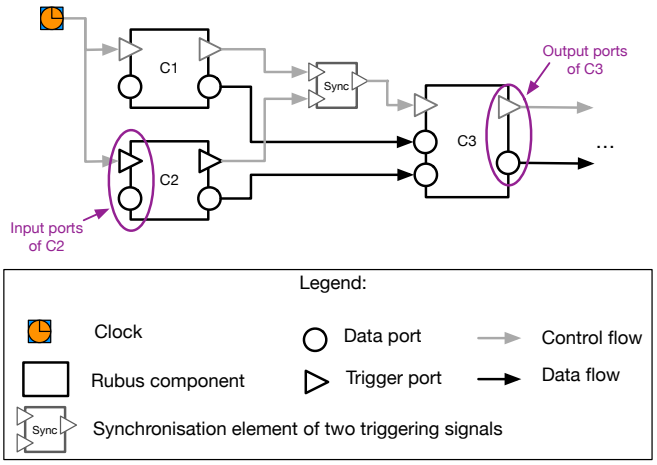


Figure 2. Three connected Rubus software components.

III. Problem Description

The run-time resources and energy usage in an embedded system can be reduced by decreasing the data produced by sensors with respect to the changing conditions in its environment. For example, a robot may require low-resolution frames to process open-space environments but may utilize high-resolution frames when analyzing close ups of detected objects. Therefore, the robot cameras may be set to provide, on-the-fly, frames with different resolutions based on, e.g., distance to the tracked objects. However, due to the rules set by the existing component models for the construction of software components, the size of a component's input data is fixed during the execution of the system. One way to ensure the guaranteed execution of the system is to allocate the system resources to software components, at the design time, to deal with the maximum footprint of data produced by sensors. For example, if a camera produces frames with 1280 x 1024 pixels, the software components that process the camera feedback utilize memory corresponding to the camera's frames. Even when the camera produces lower quality frames (e.g., 640 x 480 pixels) with a lower memory footprint, the software components are set to utilize the memory footprint characteristic to 1280 x 1024 pixel frames, resulting in under-utilization of the system memory.

We use a case study as a running example to discuss the problem in detail. The case study is centered around an underwater robot that autonomously navigates under water, fulfilling various missions (e.g., tracking red buoys) [21]. The robot contains a CPU-GPU embedded board that is connected to various sensors (e.g., cameras) and actuators (e.g., thrusters). Sensors provide a continuous flow of environment data that is processed by the GPU on-the-fly. A simplified component-based software architecture of the robot's vision system is depicted in Figure 3. The software architecture, realized using RCM, contains nine software components. The *Camera1* and *Camera2* software components are connected to the physical sensors and convert the received data into readable frames. The

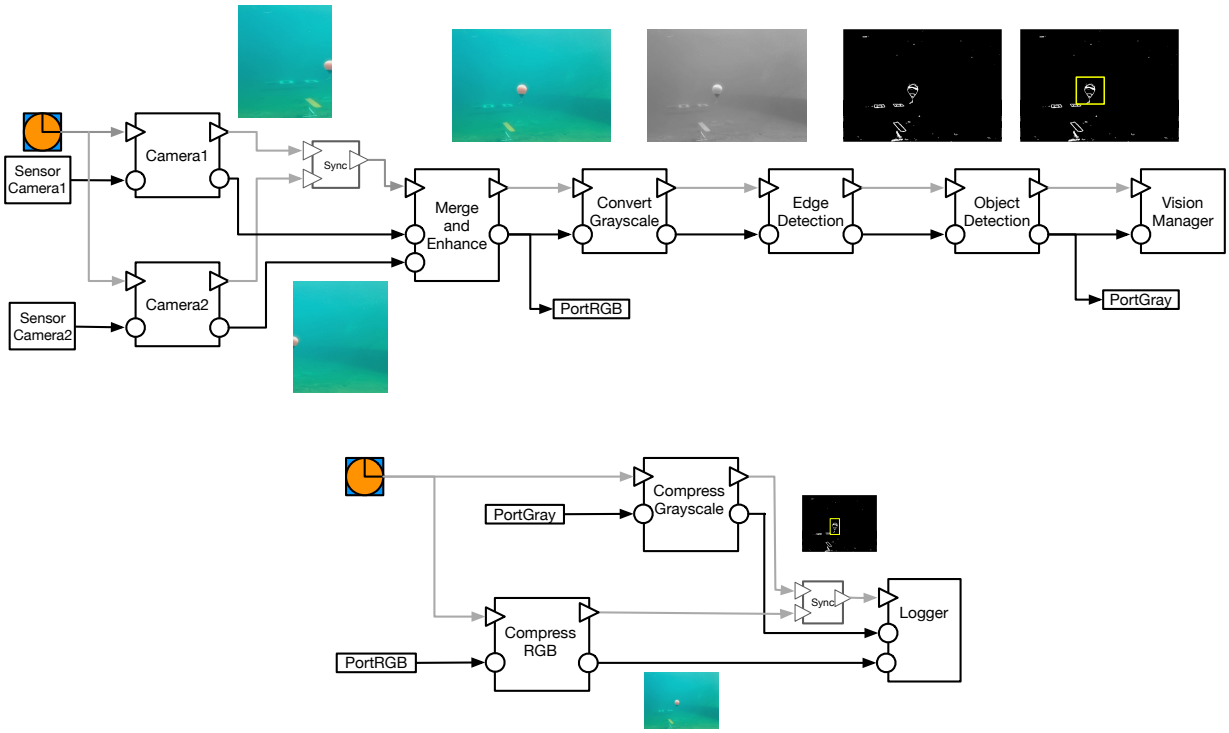


Figure 3. Component-based Rubus vision system of the underwater robot.

MergeAndEnhance software component reduces the noise and merges the two frames using the GPU. The resulted frame is converted into a gray-scale frame by *ConvertGrayscale* software component (on the GPU), which is forwarded to *EdgeDetection* software component that produces a black-and-white frame with detected edges. The *ObjectDetection* software component identifies the target object from the received frame and forwards the result to the system manager that takes appropriate actions, such as grabbing the detected objects. Due to the specifications of the functionalities (i.e., processing image), *MergeAndEnhance*, *ConvertGrayscale*, *EdgeDetection* and *ObjectDetection* components use the GPU to execute their behavior.

When the robot navigates underwater, the cameras are set to produce 640 x 480 pixel frames to track points for positioning itself. Due to the particularities of the water, sometime being muddy or the underwater vision being influenced by the weather conditions (e.g., cloudy, sunny), there is no need for high-resolution frames as the visibility is reduced. Figure 3 presents 640 x 480 pixel frames that contain several objects. While one of the missions is to track and touch buoys, the robot navigates to the detected objects. When the robot is close (e.g., one meter away) to the detected object, it requires high-resolution frames to observe and refine the details needed for the distinction between similar type of objects. In this case, cameras produce 1280 x 960 pixel frames.

Following the specifications of RCM, each software component is equipped with a constructor and a destructor. The constructor is executed once before the system run. Whereas, the destructor is executed when the system is properly switched off or reset. The constructor has the role to allocate resources needed by the software component, such as memory required by the internal behavior and output data ports. As it is executed only once, the constructor allocates a fixed memory size for the duration of entire execution life of the software component. For the presented vision system, the constructor of each software component reserves memory to handle e.g., input data of maximum size. In our running case system, the constructor of *Camera1* allocates memory space that holds 1280 x 960 pixel frames. When sensors provide frames with lower resolution and memory footprint, *Camera1* has reserved the same amount of memory (corresponding to 1280 x 960 pixel frames) from which it uses only a part, resulting in under utilization of the memory.

Besides the memory requirements, each component with GPU capability need a particular GPU computation resource (i.e., GPU thread) when processing images. For example, the *EdgeDetection* requires a number of 1280*960 GPU threads (i.e., a thread for each image pixel) to process the input image. When the camera sensors switch from 1280 x 960 to 640 x 480 pixel frames, *EdgeDetection* has reserved the same number of GPU threads corresponding to the high-resolution frame. This leads to a waste of GPU computation resources.

Another part of the underwater robot is the logger system that is composed of three software components, i.e., *CompressGrayscale*, *CompressRGB* and *Logger*. This part of

the software architecture has a non-critical functionality. The *CompressGrayscale* and *CompressRGB* components have GPU capability, that is, their image compression functionality is executed on the GPU. The purpose of this non-critical part is to compress and record various information of the robot during the underwater journey. Due to the limited (CPU and GPU) memories, the logger system is triggered by a *clock* element to save the resulting frames, with a specific (low) time frequency. These frames are copied from the RAM to a flash memory by a specific service of the operating system. If the system has more available memory then the logger system is triggered with a faster time frequency. In this case, there will be an improvement in various system activities e.g., checking the (correct) functionality of the vision system by using a higher number of processed frames. Moreover, the logger system can benefit from extra memory by delivering other system information (e.g., energy usage and temperature) that improves the debugging activity of the robot.

IV. Proposed Solution and Proof-of-concept Implementation

A. Generic Solution

In order to improve the resource utilization of non-critical parts of the embedded systems, we introduce an automatic method that, during run-time, provides information on the additional available resources that can be used by the non-critical parts. The proposed method is presented in Figure 4, which consists of four blocks: (i) Critical System, (ii) Non-critical System, (iii) Monitor, and (iv) Evaluator. The Critical System and Non-critical System blocks represent the critical and non-critical functionalities in the system as discussed in Section I. The Monitor block periodically checks (e.g., every triggering execution) the resources usage of only the Critical System. Lastly, the Evaluator block, based on the information received from the Monitor, evaluates the available system resources and provides this information to the Non-critical System. The interactions among the different blocks in Figure 4 are identified by means of the arrows. Step (arrow) 1 from Figure 4 expresses the examination of the Critical System by the Monitor. During step 2, the Monitor sends the actual usage of the resources to the Evaluator. Based on the the received information, the Evaluator has the following two options:

- if the Critical system uses as much resources as its maximum (worst case) requirement, the Evaluator informs the Non-critical system to use its default allocated resources (step 3),
- if the Critical system uses less resources than its maximum requirement, the Evaluator computes the size of the unused resources and distributes it to the Non-critical system (step 4).

B. Realization

This subsection describes the realization details of our method using the vision system case study. The first part of the section introduces groundwork details on the functionality of the component model, while the second part presents the overall realization of our method.

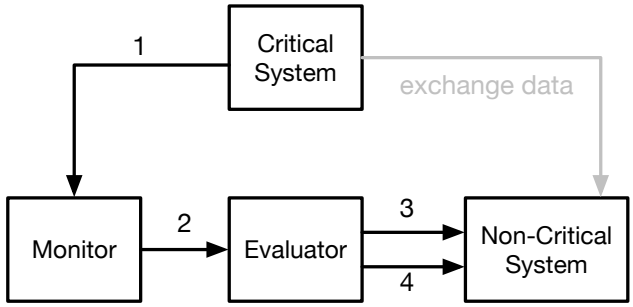


Figure 4. Overview design of the Evaluator method.

1) Component Model Functionality: Each component is characterized by a constructor and a destructor. The constructor is executed once, at the initialization of the system, and allocates as much memory as the component requires. The destructor, executed once when the system is properly switched off or reset, has the purpose to deallocate the memory. Figure 5 focuses on two connected software components from the vision system. In order to simplify the figure, we remove some of the (triggering) connections of the components. *Camera1* sends a frame to *MergeAndEnhance* component. Initially, the constructor of *Camera1* allocates memory space (on CPU memory space) to accommodate frames of maximum size (i.e., 1280 x 960 pixels). Similarly, the constructor of *MergeAndEnhance* allocates on GPU memory address, memory space for high-resolution frames (1280 x 960 pixels). Furthermore, each time when the component is executed it uses the same number of GPU threads (i.e., one thread per pixel) to process high-resolution frames. When the robot changes its mode (e.g., to save its energy) and its physical cameras send lower size frames (640 x 480 pixel frames), both *Camera1* and *MergeAndEnhance* use only a part of the memory allocated to them by their respective constructors. Moreover, *MergeAndEnhance* uses more GPU threads to process 640 x 480 pixel frames than it needs.

To send large data (i.e., larger than a scalar), components need to use pointers, as follows. The output port of *Camera1* is basically a *struct* that contains a pointer variable and two scalars, characteristics to 2D images. The port may cover other types of data, such as 3D images by including additional information, such as a third scalar. The pointer indicates to the memory address that it is at the beginning of the data to be transferred, and the two scalars (i.e., height and width) describe the size of the frame. In this way, *Camera1* passes the information (of the pointer and scalars) about the data to be transferred to the *MergeAndEnhance* component.

Figure 5 presents, in the lower part, an abstract overview of the hardware layer. In the CPU memory address section, a memory space is allocated for *Camera1* to hold 1280 x 960 pixel frames. This memory location is indicated by the

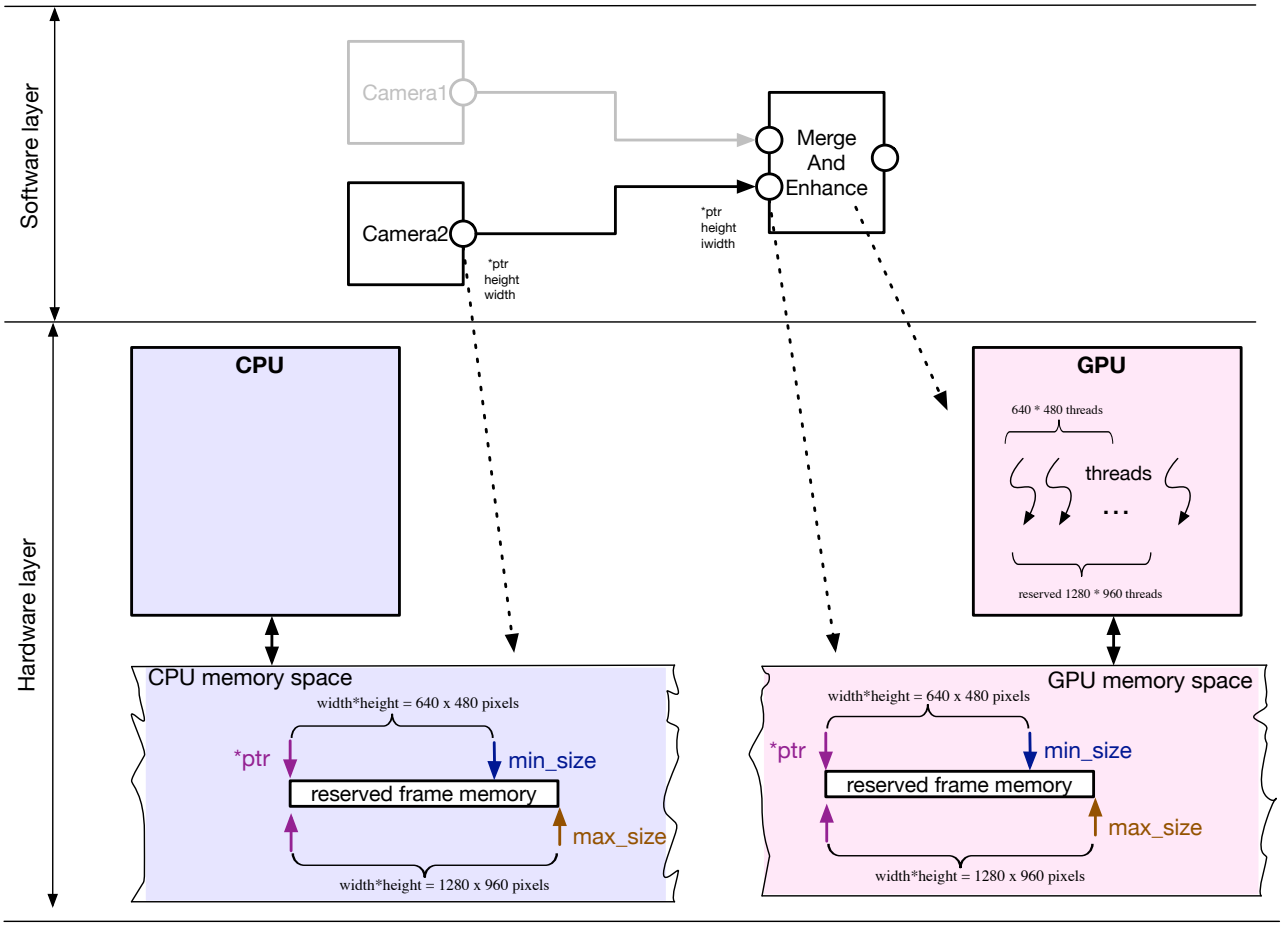


Figure 5. The requirements of two connected components.

pointer *ptr*. For frames of different resolution, *ptr* points to the same location. The difference is that for high resolution frames the size of the memory (i.e., *max_size*) is different (i.e., higher) than the size of the memory location corresponding to low-resolution frames (i.e., *min_size*). Similarly, the different sizes of memory allocation, in the GPU memory space section, corresponding to the high- and low-resolution frames in the case of *MergeAndEnhance* component are depicted in Figure 5. Furthermore, the figure presents, in an abstract way, the *MergeAndEnhance* requirements of GPU threads usage to process the frames of different sizes.

2) Vision System Realization: The vision system is composed of four parts and realized as follows.

a) The Critical System. The critical system contains the functionality that has the highest priority in the system. In our case, it produces and processes the frames, and takes decisions based on the findings. There are seven software components included in this part of the system as illustrated in Figure 6.

b) The Monitor. We realize the monitor as a service that is regularly performed by the operating system. The service checks the settings of the camera sensors and produces a value that corresponds to the frame sizes produced by the cameras, i.e., 1024 or 640.

c) The Evaluator. The evaluator is realized as a regular software component that receives its input information from the monitoring service. Because it decides the distribution of the resource memory utilized by the critical system, the priority of the *Evaluator* component is set to the highest level. Based on the value received from the monitor, the Evaluator component decides if the non-critical system can use more resources and produces the data that reflects this decision. For simplicity, the output result is a boolean variable; the output value 1 means that the non-critical system may use more resources than initially allocated, and 0 the opposite. The *Evaluator* component (i.e., its constructor, behavior function and destructor) is entirely automatically generated through our solution (see Section IV-C for details).

d) The non-critical system. The part of the system that handles the logging functionality represents the non-critical system. It has a lower priority than the critical system (and the *Evaluator* software component). It contains three software components, i.e., *CompressRGB*, *CompressGrayscale* and

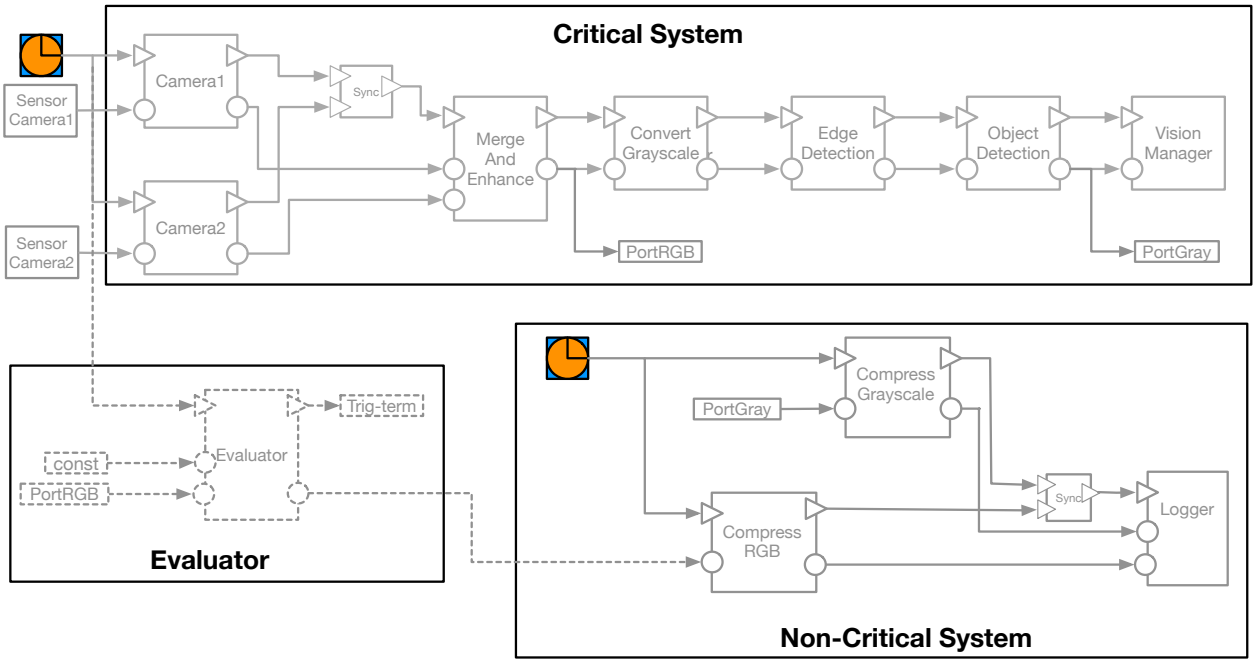


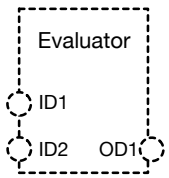
Figure 6. Realization of the Evaluator method applied on the vision system.

Logger that communicate with the *Evaluator* through an additional port. Based on the (boolean) input data received via the additional port, the non-critical components are triggered with a higher frequency rate, which leads to an improved system logger, with more compressed frames describing the underwater journey of the robot.

C. Implementation

The solution presented in this paper does not interfere with the development and execution of the critical system which is fully constructed by the developer. For the monitoring solution, we use a service provided by the OS, which is periodically executed in the background. Our solution realizes the Evaluator as a regular Rubus software component, in an automatic and transparent manner. The *Evaluator* component is generated, using the existing Rubus framework, with all its constituent parts, i.e., interface, constructor, behavior function and destructor. The interface of the component is realized as a header file, which is described in Figure 7. The interface contains two input data ports (i.e., *ID1* and *ID2*) and one output data port (i.e., *OD1*). The *ID1* port receives input data from the monitoring service, and *ID2* port receives the merged frame provided by the *MergeAndEngance* component.

The behavior function of the *Evaluator* component decides, based on the input data received from the monitoring service, to send or not the merged frame to be compressed. Figure 8 illustrates the functionality of the *Evaluator*. The output port is initialized with the image frame (line 2) received as input data, when the robot is on the low resolution mode (line 1), i.e., the Critical System does not use its entire allocated




```
1  typedef struct {
2    unsigned_char *ptr;
3    int width;
4    int height;
5  }img_format;
6
7  /* input ports */
8  typedef struct {
9    int *ID1;
10   img_format *ID2;
11 }IP_SWC_iArgs;
12
13 /* output ports */
14 typedef struct {
15   img_format OD1;
16 }OP_SWC_iArgs;
17
18 /* the interface declaration */
19 typedef struct {
20   IP_SWC_iArgs IP;
21   OP_SWC_iArgs *OP;
22 }SWC_Evaluator_iArgs;
```

Figure 7. The interface of the *Evaluator* component

memory. Although the Evaluator functionality is simple and can be easily merged to the non-critical system, we opt for the separation-of-concerns principle, which is essential in the model- and component-based software development. Moreover, the evaluator functionality can be increased to adapt to more complex systems.

The non-critical system is mostly constructed by the developer, where our approach introduces some elements that are

```
1  if( mode == Low_RESOLUTION)
2    IPA_OD1_Evaluator_ = {(void *)&
         SWC_Evaluator_iArgs->IP.ID2->ptr,
         SWC_Evaluator_iArgs->IP.ID2->width,
         SWC_Evaluator_iArgs->IP.ID2->height};
3
4  else IPA_OD1_Evaluator = {NULL, 0, 0};
```

Figure 8. Generated part of the Evaluator behavior function.

automatically generated. Initially, the non-critical system uses resources to process one frame (i.e., the grayscale frame); the constructors of *CompressGrayscale* and *Logger* components allocate memory for their functionality to compress and, respectively, log, the grayscale frame. In order to enforce a larger memory usage, the *CompressRGB* components needs to specifically allocate memory to hold the result from processing the merged frame. As the constructor is executed once at the system initialization stage, we automatically allocate memory inside the components' behavior function.

```
1  if(SWC_CompressRGB_iArgs->IP.ID1->ptr != NULL){
2        cl_mem frame_out = clCreateBuffer(context,
       CL_MEM_READ_WRITE,
       3*(SWC_CompressRGB_iArgs->IP.ID1->width) *
       (SWC_CompressRGB_iArgs->IP.ID1->height) *
       sizeof(unsigned char), NULL, NULL);
3  }
4  else {
5     IPA_OD1_CompressRGB = { NULL, 0, 0 };
6     return 0;
7  }
8
9  /* initialize parameters */
10   clSetKernelArg(kernel, 0, sizeof(cl_mem), (void
         *)&SWC_CompressRGB_iArgs->IP.ID1->ptr);
11   clSetKernelArg(kernel, 1, sizeof(int), (void *)
         &SWC_CompressRGB_iArgs->IP.ID1->width);
12   clSetKernelArg(kernel, 2, sizeof(int),(void *)&
         SWC_CompressRGB_iArgs->IP.ID1->height);
13   clSetKernelArg(kernel, 4, sizeof(cl_mem), (void
         *)&frame_out);
14
15 /* execute functionality on the input frame */
16   clEnqueueNDRangeKernel(command_queue, kernel,
         2, NULL, global_size, local_size, 0, NULL,
         NULL);
```

Figure 9. Part of the behavior function of *CompressRGB* component.

Figure 9 illustrates a section of the behavior function of the *CompressRGB* component. For the component realization, we introduce rules to generate the code from line 1 to 10. The generated code checks the input frame sent from the Evaluator (line 1). In the case that the frame exists (i.e., is not NULL), memory is specifically allocated to hold the result from processing the input merged frame (line 2). In the opposite case (i.e., the frame is NULL), the output data port is initialized with an empty frame (line 5), and the behavior function is exits (line 6). The rest of the function is defined by the component developer and is specific to the GPU functionality implemented using the OpenCL [22] syntax. Parameters that correspond to the frame specifications are set in lines 10-13, and the functionality of the component is triggered to be executed on the GPU (in line 16).

V. Evaluation

This section focuses on the evaluation of the overhead (the effect of additional elements) incurred due to the proposed solution. There are two parts that influence the overall overhead, i.e., the memory footprint and the execution time.

The memory footprint refers to the generated *Evaluator* component and the generated part of the behavior function of *CompressRGB* component (see Figure 9). The Evaluator component consists of a constructor, behavior function, and a destructor. Moreover, it has specification of its interface (i.e., ports) in a separate header file. The memory footprint of all of its code takes approximately 14 KB. We need to also add the memory size occupied by the generated parts of the *CompressRGB* component, which result in a total of 15 KB. We consider that the memory footprint overhead resulted from our approach is manageable for an embedded systems with GPUs, compared to traditional (CPU-based) embedded systems. The CPU-GPU embedded systems are characterized by a reasonable high amount of memory (i.e., order of tens of Megabyte) due to the computation power that requires high memory specifications.

TABLE I. The memory requirement of the Critical System

Component name	Memory requirement (kB)	
	Low-mode*	High-mode**
Camera1	165	307
Camera2	165	307
MergeAndEnhance	298	536
ConvertGrayscale	197	356
EdgeDetection	86	154
ObjectDetection	25	45
VisionManager	14	26
Total	950	1731

Low-mode* - camera images have 640 * 480 pixels.
High-mode** - camera images have 1280 * 960 pixels.

Besides the memory footprint, the Critical System also has a memory requirement regarding the data to be processed. Table I presents the requirement of each component of the Critical System, in the two modes of the robot. For example, *MergeAndEnhance* component, during the low-resolution mode (i.e., camera frames with 640 * 480 pixels), uses 298 kB of memory, while during the high-resolution mode, uses 536 kB of memory. During the system initialization, the Critical System is allocated, to process data, with an amount of 1731 kB of memory. When the robot switches to the low-resolution

mode, our method provides the unused 781 kB of memory of the Critical System to the Non-critical System.

Regarding the execution time, the generated *Evaluator* component may negatively affect the execution time of the critical system. In this regard, we conducted an experiment to compare the performance with and without our approach. The system on which we executed the experiments contains an embedded board AMD Accelerated Processing Unit with a Kabini architecture (i.e., CPU-GPU SoC). We used two input images, i.e., one with $640 * 480$ pixels and the other with $1280 * 960$ pixels. For each set of images, we executed two cases, one with and the other without our solution. Each case was executed 1000 times and we calculated its average execution time.

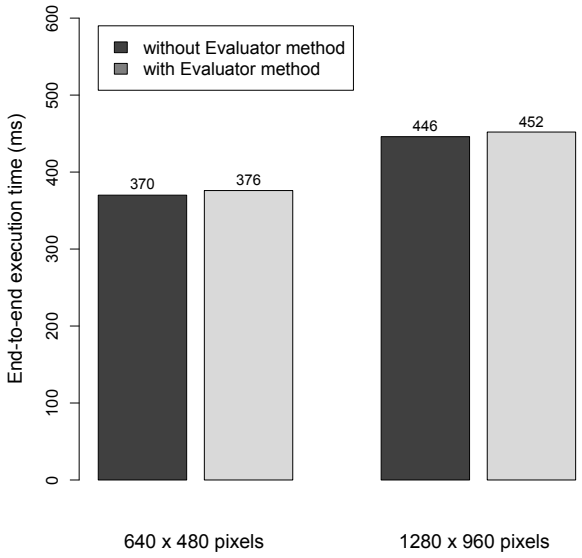


Figure 10. Usage of the Evaluator method in the vision system execution.

The results of the experiments are shown in Figure 10. A slight increase (1.3 to 1.6%) in the execution time can be observed when our solution is applied. The results indicate that the performance of the non-critical part of these systems can be significantly improved with our method at the very small execution time overhead.

Furthermore, both versions of the vision system produced the same outputs (i.e., frames). Compared to the original version, in the vision system version implemented using our approach, besides the frames produced by the *ObjectDetection* and *CompressGrayscale* components, the *CompressRGB* component also produced frames. With more data to analyze, the logger functionality of the system that is implemented using our solution showed improvement over the original version.

Besides the memory aspect evaluated in this section, our solution also deals with the GPU computation resources. The released number of GPU threads used by the critical system may be employed by the other part of the system that may use GPU simultaneously with the critical system. As the GPU allows, depending on its specifications[1], concurrent execution of several components, and by freeing an amount of GPU threads, the overall system performance may be further improved by distributing the available resources to other parts of the system that have GPU requirement.

VI. Related Work

There exist different methods to increase the memory utilization, which are presented in various surveys [23]. We mention a solution to reduce the actual allocated space for temporary arrays by using a mapping of different array parts into the same physical memory [24]. Another method proposes scratch pad memories to reduce the power consumption and improve performance [25]. These solutions are applicable at a very low level of abstraction and are not suitable to be merged with our approach, which is applicable at the implementation abstraction level where the software architecture of the application is modeled.

Regarding monitors, many works utilize them for different purposes, such as data-flow monitoring solutions to simulate large CPU-GPU systems [26], and GPU monitors for balancing the bandwidth usage [27]. An interesting work conducted by Haban et al. [28] introduces software monitors to help scheduling activities. The authors described the low overhead of the monitoring solutions, which degrade the CPU performance with less than 0.1%. In our work, we use the same type of monitors analyzed by Haban (i.e., software monitors) that have a low impact over the system performance.

In the context of developing and extending component models for embedded systems, Campeanu et al. [29] [30] introduce a solution to facilitate the component-based software development of systems with GPUs. The solution is implemented as an extension of the Rubus component model, and introduces *platform-agnostic components* with GPU capability and *adapters*. The platform-agnostic components are components that can be re-utilized on various types of platforms with GPUs, without any manual change. The adapter is a concept that facilitates the communication between components that are executed by different processing units. For example, when a CPU-executed component communicates with a GPU-executed platform-agnostic component, the adapter connects the two components and transfers, in an automatic way, the data between the CPU and GPU memory addresses. Although these two concepts were not introduced in this work, in order to mitigate the complexity of the solution, our introduced method builds upon these concepts. For example, an adapter is the artifact that transfers the data from *Camera2* to *MergeAndEnhance*, between the CPU and GPU memory spaces (see Figure 5).

Model-driven development is another paradigm adopted in the development of embedded systems. In this context, we mention the work of Rodrigues et al. that facilitate modeling

[1] e.g., an NVIDIA GPU with the Pascal architecture and compute capability 6.1 can concurrently execute up to 32 activities

of embedded systems with GPUs [31]. Due to the fact that the work has been developed in 2013, it is limited in covering the recent platform advancements. In the same context, we mention other works such as the MARTE-based framework proposed by Gamatie et al. which automatically allows generation of code for heterogeneous platforms [32].

There exists on the market various models that facilitate the development of applications with GPU capabilities. We mention the CUDA model [33], that is developed by NVIDIA vendor to specifically handle their own GPUs. CTM is another model developed by AMD to address ATI AMD GPUs [34]. OpenCL [22] is a general framework that is supported by different types of processing units (i.e., CPU, GPU, FPGA) produced by various vendors (e.g., NVIDIA, AMD, Intel). Although, in this work, we do not specifically address the functionality of the components with GPU capability such as *MergeAndEnhance* and *CompressGrayscale*, OpenCL framework was used to develop the GPU functionalities of the components used for the vision system used in the evaluation part.

VII. Conclusion

Modern embedded systems deal with huge amount of data that is originated from their interaction with the environment. GPUs have emerged as a feasible option, from the performance perspective, for processing the huge data inputs. However, with GPU-based solutions, the resource utilization remains high, which is an important aspect when dealing with resource-constrained embedded systems. In this paper, we have presented a method that improves the resource utilization for non-critical parts of CPU-GPU-based embedded systems. Whenever the critical part of the system does not fully utilize its resource requirements (i.e., memory and GPU threads) due to various reasons, such as reducing energy consumption, the presented method distributes the unused resources to the non-critical parts of the system. With the availability of more resources, the performance of the non-critical parts of the system is improved without effecting the performance of the critical parts. As a result, the performance of the overall embedded system is improved.

As a proof of concept, we have realized the method in a state-of-the-practice component model, namely the Rubus Component Model. We have also demonstrated the usability of the method using the underwater robot case study. The evaluation results indicate that the proposed method can significantly improve the performance of non-critical parts of CPU-GPU-based embedded systems at the cost of very small execution time overhead of approximately 1.5%.

An interesting future work is to extend the presented method to support the heterogeneous system architectures that include an FPGA, beside the GPU. In a system with multiple accelerators, our introduced Evaluator should be extended to decide, based on the available resources, on where the Non-critical System should execute its functionality in order to improve the overall system performance. Another future work is to model the proposed solution as a self-adaptive system that is built using the feedback control loops [35].

Acknowledgments

The work in this paper is supported by the Swedish Foundation for Strategic Research (SSF) and the Swedish Knowledge Foundation (KKS) through the projects RALF3 (IIS11-0060) and PreView respectively. We thank our industrial partners Arcticus Systems, Volvo CE and BAE Systems Hägglunds.

References

[1] G. Campeanu and S. Mubeen, “Improving run-time memory utilization of component-based embedded systems with non-critical functionality,” ICSEA, 2017, pp. 139–144.

[2] Google. Waymo - Google Self-Driving Car Project. https://waymo.com/. Accessed: 2018-02-15.

[3] NVIDIA, “NVIDIA Jetson TK1,” http://www.nvidia.com/object/jetson-tk1-embedded-dev-kit.html, accessed: 2018-02-15.

[4] AMD, “Embedded R-Series Family of Processors,” http://www.amd.com/en-us/products/embedded/processors/r-series, accessed: 2018-02-15.

[5] Samsung, “Exynos 8 Octa,” http://www.samsung.com/semiconductor/minisite/Exynos/w/solution/mod_ap/8890/, accessed: 2018-02-15.

[6] See Fast Technologies. High Speed Camera ProcImage500-Eagle. http://www.seefastechnologies.com/procimage-eng1-pi500-eagle.html. Accessed: 2018-02-15.

[7] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, “GPU computing,” Proceedings of the IEEE, vol. 96, no. 5, 2008, pp. 879–899.

[8] S. A. Manavski, “CUDA compatible GPU as an efficient hardware accelerator for AES cryptography,” 2007, pp. 65–68.

[9] T. Preis, P. Virnau, W. Paul, and J. J. Schneider, “GPU accelerated Monte Carlo simulation of the 2D and 3D Ising model,” Journal of Computational Physics, vol. 228, no. 12, 2009, pp. 4468 – 4477.

[10] J. E. Stone, J. C. Phillips, P. L. Freddolino, D. J. Hardy, L. G. Trabuco, and K. Schulten, “Accelerating molecular modeling applications with graphics processors,” Journal of computational chemistry, vol. 28, no. 16, 2007, pp. 2618–2640.

[11] T. A. Henzinger and J. Sifakis, “The embedded systems design challenge,” in International Symposium on Formal Methods. Springer, 2006, pp. 1–15.

[12] I. Crnkovic and M. P. H. Larsson, Building reliable component-based software systems. Artech House, 2002.

[13] D. Brugali and P. Scandurra, “Component-based robotic engineering (part i)[tutorial],” IEEE Robotics & Automation Magazine, vol. 16, no. 4, 2009, pp. 84–96.

[14] D. Box, Essential COM, 1st ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997.

[15] R. Monson-Haefel, Enterprise JavaBeans. Sebastopol, CA, USA: O’Reilly & Associates, Inc., 1999.

[16] I. Crnkovic, S. Sentilles, A. Vulgarakis, and M. R. Chaudron, “A classification framework for software component models,” IEEE Transactions on Software Engineering, vol. 37, no. 5, 2011, pp. 593–615.

[17] “AUTOSAR - Technical Overview,” http://www.autosar.org, accessed: 2018-02-15.

[18] K. Hanninen, J. Maki-Turja, M. Nolin, M. Lindberg, J. Lundback, and K.-L. Lundback, “The rubus component model for resource constrained real-time systems,” in Industrial Embedded Systems, 2008. SIES 2008. International Symposium on. IEEE, 2008, pp. 177–183.

[19] I. Application, “Implementation of IEC 61131-3,” Geneva: IEC, 1995.

[20] S. Sentilles, A. Vulgarakis, T. Bures, J. Carlson, and I. Crnkovic, "A Component Model for Control-Intensive Distributed Embedded Systems," in 11th International Symposium on Component Based Software Engineering (CBSE), vol. 8. Springer, October 2008, pp. 310–317.

[21] C. Ahlberg, L. Asplund, G. Campeanu, F. Ciccozzi, F. Ekstrand, M. Ekstrom, J. Feljan, A. Gustavsson, S. Sentilles, I. Svogor et al., "The Black Pearl: An autonomous underwater vehicle," 2013.

[22] KHRONOS Group, "The OpenCL specifications," https://www.khronos.org/developers/reference-cards/, accessed: 2018-02-15.

[23] P. R. Panda, F. Catthoor, N. D. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandercappelle, and P. G. Kjeldsberg, "Data and memory optimization techniques for embedded systems," ACM Transactions on Design Automation of Electronic Systems (TODAES), vol. 6, no. 2, 2001, pp. 149–206.

[24] M. A. Miranda, F. V. Catthoor, M. Janssen, and H. J. De Man, "High-level address optimization and synthesis techniques for data-transfer-intensive applications," IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 6, no. 4, 1998, pp. 677–686.

[25] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel, "Scratchpad memory: design alternative for cache on-chip memory in embedded systems," in Proceedings of the tenth international symposium on Hardware/software codesign. ACM, 2002, pp. 73–78.

[26] B. R. Bilel, N. Navid, and M. S. M. Bouksiaa, "Hybrid CPU-GPU distributed framework for large scale mobile networks simulation," in Proceedings of the 2012 IEEE/ACM 16th International Symposium on Distributed Simulation and Real Time Applications. IEEE Computer Society, 2012, pp. 44–53.

[27] M. K. Jeong, M. Erez, C. Sudanthi, and N. Paver, "A QoS-aware memory controller for dynamically balancing GPU and CPU bandwidth use in an MPSoC," in Proceedings of the 49th Annual Design Automation Conference. ACM, 2012, pp. 850–855.

[28] D. Haban and K. G. Shin, "Application of real-time monitoring to scheduling tasks with random execution times," IEEE Transactions on software engineering, vol. 16, no. 12, 1990, pp. 1374–1389.

[29] G. Campeanu, J. Carlson, and S. Sentilles, "Developing CPU-GPU embedded systems using platform-agnostic components," in Software Engineering and Advanced Applications (SEAA), 2017 43rd Euromicro Conference on. IEEE, 2017, pp. 176–180.

[30] G. Campeanu, J. Carlson, S. Sentilles, and S. Mubeen, "Extending the Rubus component model with GPU-aware components," in Component-Based Software Engineering (CBSE), 2016 19th International ACM SIGSOFT Symposium on. IEEE, 2016, pp. 59–68.

[31] A. W. O. Rodrigues, F. Guyomarc'H, and J. L. Dekeyser, "An MDE approach for automatic code generation from UML/MARTE to OpenCL," Computing in Science & Engineering, vol. 15, no. 1, 2013, pp. 46–55.

[32] A. Gamatie, S. Le Beux, E. Piel, R. Ben Atitallah, A. Etien, P. Marquet, and J.-L. Dekeyser, "A model-driven design framework for massively parallel embedded systems," ACM Trans. Embed. Comput. Syst., 2011, pp. 39:1–39:36.

[33] "NVIDIA CUDA C programming guide," https://docs.nvidia.com/cuda/cuda-c-programming-guide/, accessed: 2018-02-15.

[34] J. Hensley, "AMD CTM overview," in ACM SIGGRAPH 2007 courses. ACM, 2007, p. 7.

[35] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," Computer, vol. 36, no. 1, Jan 2003, pp. 41–50.