Improving Run-Time Memory Utilization of Component-based Embedded Systems with Non-Critical Functionality

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

***Abstract*—Many contemporary embedded systems have to deal with huge amount of data, coming from the interaction with the environment, due to their data-intensive applications. However, due to some inherent properties of these systems, such as limited energy and resources (compute and storage), 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. Component-based software development techniques and models have proven to be efficient for the development of these systems. Many component models used in the industry (e.g., Rubus, IEC 61131) allocate, at the system initialization, enough resources to satisfy the demands of the system's critical functionality. These resources are retained by the critical functionality even when they are not fully utilized. In this paper, we introduce a method that, when possible, distributes the unused memory of the critical functionality to the non-critical functionality in order to improve its performance. The method uses a monitoring solution that checks the memory utilization, and triggers the memory distribution whenever possible. As a proof of concept, we realize the proposed method in an industrial component model. As an evaluation, we use an underwater robot case study to evaluate the feasibility of the proposed solution.**

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

I. INTRODUCTION

Embedded systems are found in almost all electronic products that are available today. These systems find their applications in a vast range of systems, i.e., from small-sized devices, such as watches and telephones to large-sized systems, such as cars and airplanes. Many modern embedded systems process huge amount of data that is originated from their interaction with the environment. One example is the Google autonomous car that processes around 750 MB data per second [1]. The reduced computation power and sequential execution of software that characterize many embedded systems can represent a challenge to deliver the performance level required by the systems when processing huge amount of data.

Graphics Processing Units (GPUs) represent a solution to deliver the required performance level when the system deals with processing huge amount of data. Characterized by a parallel execution model, the GPU can process multiple data in parallel. An aspect of the GPU is that it cannot function without a CPU; considered as the brain of the system, the CPU triggers all GPU-related activities (e.g., parallel execution of functions). The latest technological developments allow the combination of CPUs and GPUs on the same embedded boards, resulting in various heterogeneous platforms, such as NVIDIA Jetson [2] and AMD R-464L [3].

Due to the specifics of embedded systems, such as limited compute and memory resources, the amount of data captured from the environment can significantly impact the management of the system resources while delivering the required performance. One way to optimize the resource usage is to collect variable stream-size of data from the sensors depending upon different environment situations. For example, camera sensors (e.g., ProcImage500-Eagle [4]) with configurable resolutions may provide: *i)* frames with high resolution, and *ii)* frames with low resolution. While the high resolution frames require larger memory footprints and more computation power (and energy) to be processed (on GPUs), the low resolution frames are delivered with faster frame rate, occupy less memory and require lower computation power for GPU processing. Depending on the environment circumstances, cameras may provide high or low quality frames. For example, a robot fitted with such a camera may use low resolution data frames to examine its position. On the other hand, the robot may use high resolution frames to inspect the target objects in a detailed manner.

The system resources (e.g., memory and computation power) in many embedded systems are shared between the critical (with real-time requirements) and non-critical functionality. The goal in the case of the critical functionality is to meet all the timing requirements. Whereas, the best-effort service is targeted in the case of the non-critical functionality. Hence, the system needs to ensure that all the required resources are always available to the critical part of the application. For example, a vision system of a robot represents critical functionality. 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 lower-resolution frames, the system still occupies the same amount of resources as if it were processing the high-resolution frames. As a result, the system resources are wasted when the critical functionality does not need them. In our point of view, the non-critical functionality can benefit from these resources in the intervals where they are not used by the critical functionality. For example, when the robot utilizes lower resolution frames, a logger system (non-critical functionality) would benefit from extra memory (not being used by the vision system) to save more information about the system activities.

In order to deal with the complexity, among other challenges, the software for embedded systems is developed using the principles of Component-Based Software Engineering (CBSE) and Model-Based Engineering [5] [6]. Using these principles, models are used throughout the development process and the software is constructed by connecting reusable

 ISBN: 978-1-61208-590-6

software units, called the software components. CBSE and MBD have been successfully adopted by the industry through component models, such as AUTOSAR [7], Rubus Component Model (RCM) [8] and IEC 61131 [9]. The existing component models that can be used to build stream-of-event applications (e.g., RCM, AUTOSAR, IEC 61131 and ProCom[10]), face a challenge to deal 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).

This paper provides an automatic method to compute the unused resources of the critical part of the system, and distribute them to the (non-critical) parts of the system. This is achieved by using a monitoring solution that monitors the critical part of the system and detects when it changes its resource requirements. After detection, the monitoring solution triggers our proposed method that calculates the unused memory, based on the actual resource usage of the critical system. This information is passed to the (non-critical) part of the system that can benefit from utilizing the freed resources.

The rest of the paper is organized as follows. Section II describes the background and related work. 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 and its realization is presented in Section V. Section VI discusses the implementation of the solution. The evaluation of our method applied to the case study is discussed in Section VII. Finally, Section VIII concludes the paper.

II. Background and Related Work

GPUs were developed in 90s and were employed only in graphic-based applications. By time, due to the increase in their computation power and ease of use, GPUs have been utilized in different type of applications, becoming the general-purpose processing units referred to as GPGPUs [11]. For example, cryptography applications [12] and Monte Carlo simulations [13] have 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 [14].

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 latest technological developments allow various vendors, such as NVIDIA, Intel, AMD and Samsung to combine CPUs and GPUs on the same embedded board. For example, there are boards known as System-on-Chips (SoCs) that merge together CPUs and GPUs onto the same physical chip, such as NVIDIA Jetson TK1 [2] and Samsung Exynos 8 [15].

Regarding embedded systems that contain GPUs, there are model- and component-based software engineering extensions to facilitate the development of CPU-GPU applications [16] [17]. Component models follow various interaction styles that are suitable for different types of applications [18]. We mention the *request-response* and *sender-receiver* interaction styles that are utilized in AUTOSAR component model when developing automotive applications. Another style utilized by e.g., Rubus and IEC 61131 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.

There exist different methods to increase the memory utilization, which are presented in various surveys [19]. 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 [20]. Another method proposes scratch pad memories to reduce the power consumption and improve performance [21]. 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 [22], and GPU monitors for balancing the bandwidth usage [23]. An interesting work conducted by Haban et al. [24] introduced 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.

III. Problem

One way to reduce resource and energy usage of embedded systems is to decrease the data produced by sensors with respect to e.g., environment conditions. 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 tracked objects.

Due to the rules that existing component models apply 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) [25]. The robot contains a CPU-GPU embedded board that is connected to various sensors (e.g., cameras) and actuators (e.g., thrusters).

 ISBN: 978-1-61208-590-6

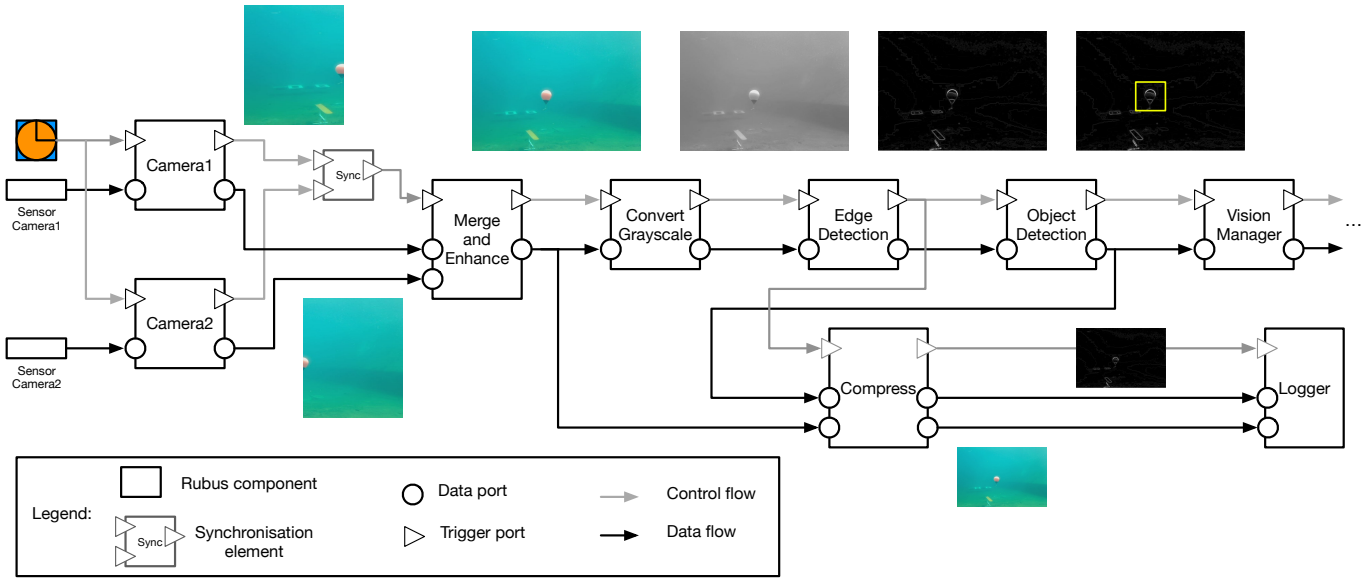

Figure 1. Component-based Rubus vision system of the underwater robot.

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 1. 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 *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.

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 1 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., 1 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 execution, while the destructor is executed when the system is properly switched off or reset. The constructor has the role to allocate resources needed by the 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 component. For the presented vision system, the constructor of each 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.

Another part of the underwater robot is the logger system that is composed of two software components, i.e., *Compress* and *Logger*. This part of the software architecture has a non-critical functionality. 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 memory (RAM), *Compress* and *Logger* software components save the resulted frames (onto RAM) from *ObjectDetector* software component. These frames are copied from the RAM to a flash memory by a specific service of the operating system. If more memory was available to the logger system, it would have also saved the unaltered (original) frames from the *MergeAndEnhance* software component. This would improve various system activities e.g., checking the (correct) functionality of the vision system by comparing the original and processed frames. Moreover, the logger system may benefit from extra memory by delivering other system information (e.g., energy usage and temperature) that improves the debugging activity of the robot.

IV. 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. Figure 2 presents the overview design of our proposed method and its interactions with the critical and non-critical parts of embedded system.

Our method uses a monitoring solution that periodically checks (e.g., every execution) the memory usage of the critical system. Step (arrow) 1 from Figure 2 expresses the examina-

 ISBN: 978-1-61208-590-6

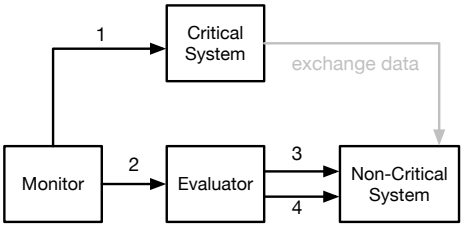

Figure 2. Overview design of the proposed method.

tion of the critical system by the monitoring solution. During step 2, the monitor sends the actual memory usage to the evaluator. Based on the the received information, the evaluator has two following two options.

- If the critical system uses as much memory as its maximum (worst case) requirement, the evaluator informs the non-critical system to use its default memory allocated memory (step 3).
- If the critical system uses less memory than its maximum requirement, the evaluator computes the size of the unused memory and distributes it to the non-critical system (step 4).

V. Realization

This section 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.

A. 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, executing once when the system is properly switched off, has the purpose to deallocate the memory. Figure 3 shows two connected software component from the vision system. In order to simplify the figure, we remove some of the (triggering) connections to the component. Camera1 sends a frame to MergeAndEnhance component. Initially, the constructor of Camera1 allocates memory space to accommodate frames of maximum size (i.e., 1280 x 960 pixels). When the robot changes its mode (e.g., for saving its energy) and its physical cameras send lower size frame (i.e., 640 x 480 pixels), Camera1 uses only a part of the memory, which was allocated by its constructor.

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 (from RAM) to be transferred to the MergeAndEnhance component. We can see in the figure that the transferred data is a frame of 640 x 480 pixels, which means that there is some unused memory. Using this information (i.e., size), the Evaluator

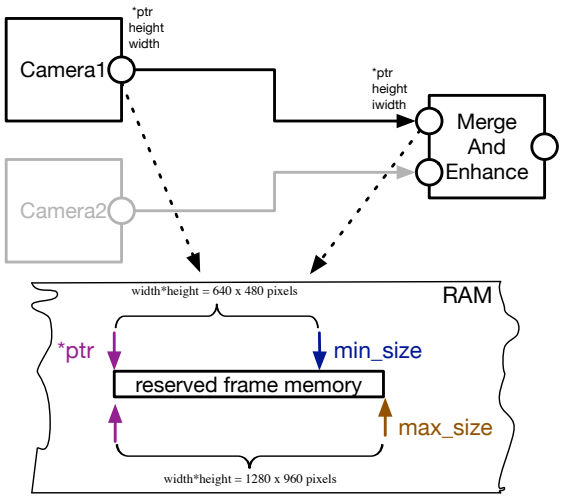

Figure 3. Data transferring between two components.

component calculates the total unused memory of the vision system and inform the Logger system to use it.

B. 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 4.

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 this value, 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.

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 evaluator software component. It contains two software components, i.e., *Compress* and *Logger* that communicate with the *Evaluator* through an additional port. Based on the (boolean) data received via the additional port, the two non-critical components use one or two frames in their computations.

VI. Implementation

The solution presented in this paper does not interfere with the development and execution of the critical system. It is con-

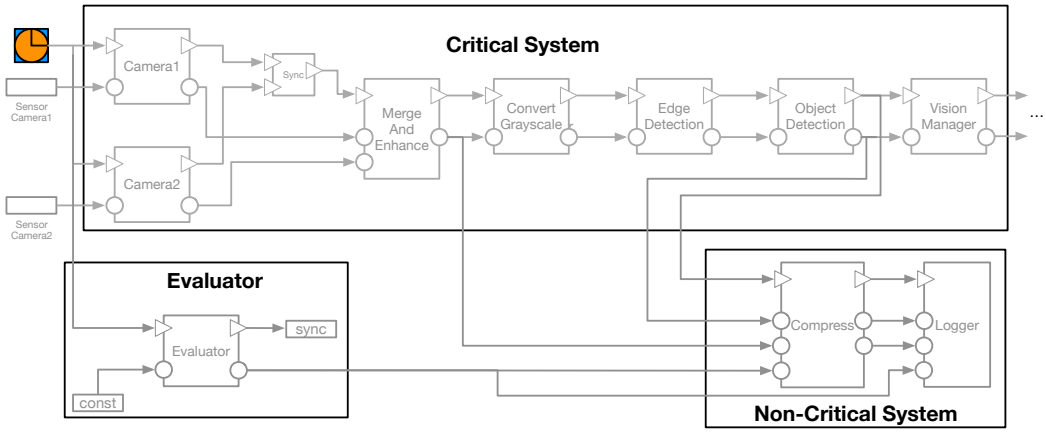

Figure 4. Realization of the proposed method applied on the vision system.

structed by the developer. For the monitoring solution, we use a service provided by the OS. The evaluator is implemented as a regular component with an input and output data port. Through the input port, it receives data from the monitoring solution, while the output port provides a boolean data. At this stage of our solution, the functionality is simple and decides, based on the input value, if the non-critical system can have access to more resources or not. Although the 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 for more complex systems.

```
1  if(<InPort3.Name>==1)
2  {
3    cl_mem frame2_out = clCreateBuffer(context, CL_MEM_READ_WRITE,
         3*(<OutPort2.Name>->width)*(<OutPort2.Name>->height) *
         sizeof(unsigned char), NULL, NULL);
4
5  /* initialize parameters */
6    clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&
         <InPort2.Name>->ptr);
7    clSetKernelArg(kernel, 1, sizeof(int), (void *)&<InPort2.Name>->
         width);
8    clSetKernelArg(kernel, 2, sizeof(int),(void *)&<InPort2.Name>->
         height);
9    clSetKernelArg(kernel, 4, sizeof(cl_mem), (void *)&frame2_out);
10
11 /* execute functionality on the second frame */
12   clEnqueueNDRangeKernel(command_queue, kernel, 2, NULL,
         global_size, local_size, 0, NULL, NULL);
13 }
```

Figure 5. Generated part of the behavior function.

The non-critical system is mostly constructed by the developer, where our approach introduces some elements that are automatically generated. Initially, the non-critical system uses resources to process one frame; the constructors of *Compress* and *Logger* components allocate memory for one frame to be used in their functionality. In order to enforce a larger memory usage, the two components need to allocate more memory to hold the result from processing the second frame. As the constructor is executed once at the system initialization stage, we automatically allocate memory inside the components' behavior function.

Figure 5 illustrates the code generated inside the behavior function of each software component from the non-critical system. We assume that each port has a name. For simplicity, all the components from the non-critical system have an input port with a boolean value (i.e., 1 and 0) that is connected to the *Evaluator* component. Line 1 checks the value sent from the Evaluator, where 1 means that the component can use additional memory to process the second frame. In line 3, memory is allocated to hold the result from processing the second frame. Specific to the GPU functionality implemented using the OpenCL syntax, parameters that correspond to the second frame specifications, are set in lines 6-9. Finally, the same functionality that processes the first frame is applied to the second frame, in line 12.

VII. Evaluation

As our approach introduces additional elements to the system, this section focuses on the evaluation of overhead 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 each behavior function of the non-critical system (see Figure 5). 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 *Compress* and *Logger* components, 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.

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

 ISBN: 978-1-61208-590-6

two cases, one with and the other without our solution. Each case was executed 100 times and we calculated its average execution time.

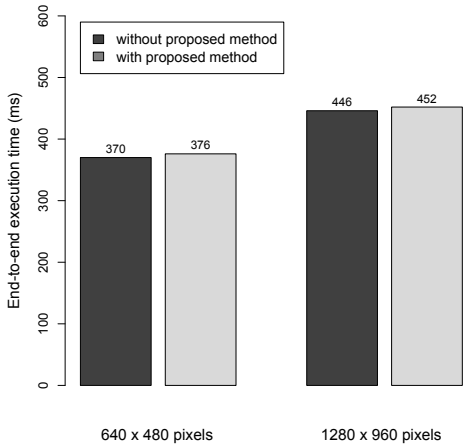

Figure 6. Usage of the proposed method in the vision system execution.

The results of the experiments are shown in Figure 6. 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.

VIII. 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 required memory due to various reasons, such as reducing energy consumption, our method distributes the unused memory to the non-critical part of the system that can use the resources to improve its performance. As a proof of concept, we have realized the method in a state-of-the-practice 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 performance of the non-critical part of CPU-GPU-based embedded systems can be significantly improved with our method at the very small execution time overhead of approximately 1.5%.

Acknowledgments

The work in this paper has been supported by the RALF3 project - (IIS11-0060) through the Swedish Foundation for Strategic Research (SSF).

References

[1] Google. Waymo - Google Self-Driving Car Project. https://waymo.com/. Retrieved: July, 2016.

[2] NVIDIA, "NVIDIA Jetson TK1," http://www.nvidia.com/object/jetson-tk1-embedded-dev-kit.html, retrieved: July, 2017.

[3] AMD, "Embedded R-Series Family of Processors," http://www.amd.com/en-us/products/embedded/processors/r-series, retrieved: July, 2017.

[4] See Fast Technologies. High Speed Camera ProcImage500-Eagle. http://www.seefastechnologies.com/procimage-eng1-pi500-eagle.html. Retrieved: July, 2016.

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

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

[7] "AUTOSAR - Technical Overview," http://www.autosar.org, retrieved: July, 2017.

[8] 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.

[9] I. Application, "Implementation of IEC 61131-3," Geneva: IEC, 1995.

[10] 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.

[11] 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.

[12] S. A. Manavski, "CUDA compatible GPU as an efficient hardware accelerator for AES cryptography," in IEEE International Conference on Signal Processing and Communications. ICSPC 2007.

[13] 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.

[14] J. E. Stone, J. C. Phillips, P. L. Freddolino, D. J. Hardy, and L. G. Trabuco, "Accelerating molecular modeling applications with graphics processors," Journal of computational chemistry, 2007.

[15] Samsung, "Exynos 8 Octa," http://www.samsung.com/semiconductor/minisite/Exynos/w/solution/mod_ap/8890/, retrieved: July, 2017.

[16] G. Campeanu, J. Carlson, and S. Sentilles, "Component allocation optimization for heterogeneous cpu-gpu embedded systems," in Software Engineering and Advanced Applications (SEAA), 2014 40th EUROMICRO Conference on. IEEE, 2014, pp. 229–236.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

 ISBN: 978-1-61208-590-6