A 2-Layer Component-based Architecture for Heterogeneous CPU-GPU Embedded Systems

Gabriel Campeanu and Mehrdad Saadatmand

[1] Mälardalen Real-Time Research Center (MRTC),
Mälardalen University, Västerås, Sweden
{gabriel.campeanu,mehrdad.saadatmand}@mdh.se
[2] SICS Swedish ICT, Västerås, Sweden
mehrdad@sics.se

Abstract. Traditional embedded systems are evolving into heterogeneous systems in order to address new and more demanding software requirements. Modern embedded systems are constructed by combining different computation units, such as traditional CPUs with Graphics Processing Units (GPUs). Adding GPUs to conventional CPU-based embedded systems enhances the computation power but also increases the complexity in developing software applications. A method that can help to tackle and address the software complexity issue of heterogeneous systems is component-based development.
The allocation of the software application onto the appropriate computation node is greatly influenced by the system information load. The allocation process is increased in difficulty when we use, instead of common CPU-based systems, complex CPU-GPU systems.
This paper presents a 2-layer component-based architecture for heterogeneous embedded systems, which has the purpose to ease the software-to-hardware allocation process. The solution abstracts the CPU-GPU detailed component-based design into single software components in order to decrease the amount of information delivered to the allocator. The last part of the paper describes the activities of the allocation process while using our proposed solution, when applied on a real system demonstrator.

1 Introduction

Due to the advances in micro and nano technology fabrication, traditional embedded systems are becoming more and more complex. The traditional homogeneous unicore CPU-based systems have emerged as heterogeneous systems which combine various processing units, such as multi-core CPUs and GPUs. The GPU computation power brings considerable speed-ups compared to the traditional CPU, for various software applications such as n-body simulations [11] or 3D reconstruction medical systems [13]. Also, the large GPU parallel computation power made possible the appearance of new and more complex system applications such as vehicle vision systems [7] and autonomous vision-based robots [9]. The combination of CPU and GPU increases the system computation power,

but comes with a cost: being a different computation unit with its own memory system, the GPU increases the complexity of the software system.

A way to tackle the newly increased software complexity is through component-based development (CBD). CBD increases developers productivity by constructing complex software applications out of existing software blocks know as components. Several advantages arrive by following this development approach; among them, we can mention an increased productivity and a faster time-to-market.

Developing the component-based design of a system which requires to use the heterogeneous CPU-GPU hardware is demanding due to the variety of component candidates. From a repository of CPU and GPU-based components, different combinations of components with their respective properties can construct different alternatives that have the same functionality. In an embedded system with multiple heterogeneous hardware nodes, the software-to-hardware allocation process needs to consider all component-based design alternatives[3] in order to choose the right one w.r.t. the system properties and constraints (e.g., performance optimization). For instance, having a system design composed of only GPU-based components may result in an infeasible allocation scheme due to constraints such as the GPU hardware resource limitation.

Determining the software-to-hardware allocation is an NP-hard problem [5]. This challenge is increased even more when instead of using common CPU-based systems, we use complex CPU-GPU based platforms. In our previous work [6], we developed a software component allocation model for heterogeneous CPU-GPU systems. We used the CBD approach to model the software system, and characterize it with extra-functional properties (e.g., CPU and GPU memory usage). Using the allocation model, we constructed a semi-automatic allocator which balances the hardware resource usage and optimizes the system performance. The work does not consider multiple alternatives in its allocation process. This negatively influences the outcome of the allocation process because it excludes feasible solutions.

Having multiple alternatives increases the information load of the allocator. For example, the allocator needs to take in consideration each component properties (e.g., memory, CPU and GPU usage) and the information regarding the communication links between the connected components from each alternative. Due to the tightly-connected nature of the GPU to the CPU, the allocator also requires to fulfill the constraint of deploying the entire variant onto a single heterogeneous processing node in order to not negatively influence the overall system performance.

In this paper, we propose a 2-layer architecture to ease the software-to-hardware allocation. Both of the layers describe the same system that is using GPU computation power to fulfill its functionality. The first layer contains the alternatives and their detailed properties such as communication links or memory usage. We propose a second layer that encapsulates the alternatives into a software component with multiple variants. Each variant is characterized by a

[3] for the rest of the paper, the "component-based design alternative" term will be simply referred as "alternative"

distinctive set of properties that reflects the attributes of all of the components contained by its corresponding alternative. For computing the allocation scheme, we use the properties of the abstracted second layer. After a suitable component variant is selected by the allocator, we return to the first layer in order to describe the full details of the selected alternative which was initially abstracted away. For future adjustments of the system e.g., to be used in other contexts, the first layer provides the needed component detailed view to the developer. Using two distinct levels of granularity, decreases the information load and constraints on the allocator, making the allocation process easier. Another advantage that follows from our approach is that it can improve the allocation scalability. More complex systems (e.g., number of components or properties) may be handled now by the allocator using the 2-layer approach, when the information load of all possible allocation scenarios is decreased.

The rest of the paper is organized as follows. Section 2 describes, using a running example, the software-to-hardware allocation challenges when dealing with many alternatives. The details of our solution are presented in Section 3. A case study is implemented in Section 4 in order to describe the feasibility and benefits of our solution. Related work is described by Section 5 while Section 6 presents the paper conclusion and future work.

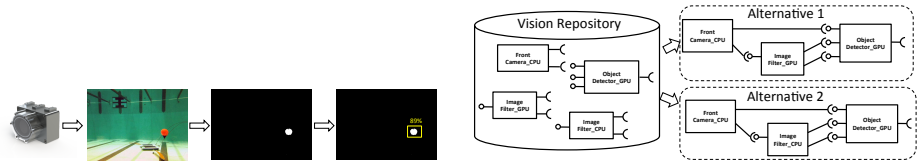
2 A CPU-GPU component-based design

A component-based application with heterogeneous computing uses a hardware platform which contains several computation nodes with different architectures, such as multi-core CPU and GPU. Due to its parallel processing power, GPUs bring as benefits an increased computation diversity and power. The disadvantage of using, adjacent to CPU, a different processing unit with its own architecture and memory system, reflects in an increased complexity of the software application. Another drawback is that the GPU can not be used independently of the CPU. Considered as the brain of the system, the CPU is the one that triggers all the GPU specific operations, such as data transfer activities between the main RAM memory and GPU memory system. Hence, there is a high communication between the two processing units. In a multi-node system, the connected CPU and GPU-based components are desired to be placed on the same heterogeneous computation node to not negatively affect the total system performance.

We consider that a repository contains two types of components; one type requires only the CPU for its functionality, and another uses also GPU to fulfill its functionality. For a specific function, we may have both types of components in the repository as different version implementations of the respective function. For example, an image processing component may have three component versions in the repository, one that uses only the CPU and the other two that uses also the GPU. The components that use the GPU, may have a different usage of the GPU resources, hence, their properties are different, e.g., in GPU memory and computation threads usage. From a repository populated with CPU and GPU software components, by having different component combinations (e.g.,

CPU-GPU, CPU-GPU-GPU), several potential alternatives with the same functionality can be constructed. In a complex system with multiple heterogeneous hardware nodes, an alternative composed of only GPU-based components may not be a feasible allocation solution due to the hardware resource limitations. Hence, the allocation process should consider all the alternatives in its allocation activity.

In the following example, we describe a component-based design for a demonstrator with heterogeneous embedded CPU-GPU hardware. The demonstrator, an underwater robot, is developed at Mälardalen University, Sweden [2]. The purpose of the demonstrator is to autonomously navigate under water in e.g., tracking various objects, using its vision system that contains two (front and bottom) cameras.



(a) The vision frame activities (b) Two vision alternatives

Fig. 1: Data activities and alternatives of the front vision system

To develop the demonstrator's front vision system, CPU and GPU software components are used from the system repository, as follows. In our example, a simplified vision system is composed of three components. The *FrontCamera* component communicates with the physical camera and receives frames from the underwater environment. The frames are forwarded to the *ImageFilter* component which processes and converts them into black-and-white frames. The *ObjectDetector* component analyzes the filtered frames to detect various objects (e.g., red buoy). Figure 1a describes the data flow activity of the front vision system, using an existing underwater frame from the demonstrator front camera.

Only the *ImageFilter* and *ObjectDetector* benefit from utilizing a GPU due to nature of their functionality, i.e., image processing. For *ImageFilter* component, the repository contains two versions, i.e., a CPU and GPU-based component, each with different properties. Using CPU-GPU combinations of the repository's components results in different vision alternatives. Figure 1b depicts the alternatives of the front vision system based on the repository content. In our case, for image processing, GPU provides a better performance than the CPU due to its massive parallel processing power. The alternative that contains two GPU-based components (i.e., *ImageFilter_GPU* and *ObjectDetector_GPU*), has a high usage of the GPU (e.g, GPU memory and computation threads) and a good performance. The alternative which contains only one GPU-based component, uses less the GPU resources and has a lower performance compared to the previous alternative.

For our small example, all the front vision system alternatives have the same simple design, i.e., three connected components. In a more advanced application, the vision system may have different design for different alternatives. For example, an alternative may contain three GPU-based components, while another may have two GPU-based components and two CPU-based components.

In general, considering all possible alternatives and selecting one best fit to be allocated on a heterogeneous hardware node, brings an information load that may hurt the efficiency of the allocator. The allocator needs to take in consideration each component properties from each variant and to have a component-to-variant mapping. Also, because of the CPU-GPU dependency, the GPU-based components are desired to be deployed with their connected CPU-based components, otherwise e.g., a high communication between them may negatively influence the system overall performance. The complexity of the allocation process is increased by the number of the alternatives and their content components, which may affect the allocator performance (e.g., scalability, allocation time).

3 Solution overview

To ease the software-to-hardware allocation process, we propose a 2-layer architecture view for the component-based design of the system. The first layer contains the details of the system alternatives. We propose to abstract away the complexity of these system alternatives, by using an abstracted second layer. This second layer compacts a system alternatives into a single component with many variants. A component variant is characterized by properties that reflect the components' properties of the corresponding alternative.

Figure 2 presents a component with two variants of the front vision system. The figure illustrates only one alternative which is composed of three components. The elements of the alternative, i.e., components and communication links, are characterized by different properties such as RAM memory usage or bandwidth. Each variant hides away the full details of its corresponding alternative, and exposes the overall properties of the abstracted structure.

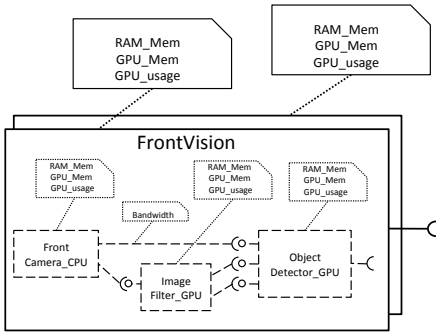


Fig. 2: Two front vision alternatives abstracted to a component with two variants

The allocator, using the abstracted layer properties, chooses the fittest component variant w.r.t. the rest of the system properties and other constraints. Once a component variant is selected, the system is described by the first layer where the alternative that corresponds to the selected component variant is displayed. Figure 3 presents the steps of our approach. From the components contained by the repository, several alternatives for different systems are constructed in step 1. These may be constructed either manually by the system developer, or automatically. In step 2, all alternatives with the same functionality are compacted into a multi-variant component. Although not described in the figure, at this step, the alternatives properties are synthesized into the properties of the component variants. The allocator, which is an automatic system, receives in step 3, the properties of the multi-variant components. Using also other information (e.g., software and hardware models, constraints), the allocator selects, during step 4, the fittest component variants for the system allocation scheme. Once the variants are selected, the last step exposes the detailed alternatives abstracted by the selected variants.

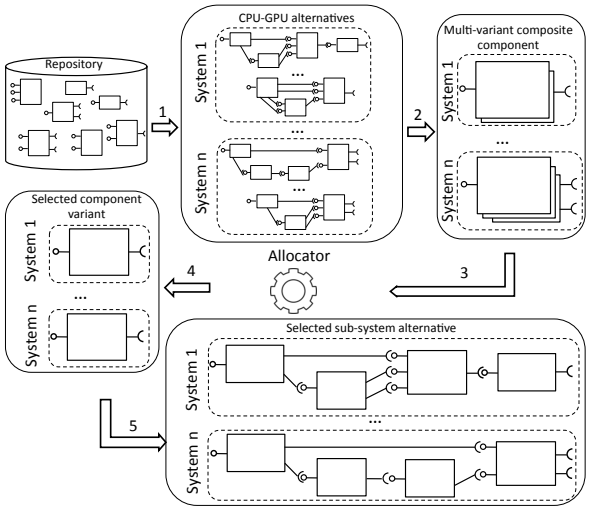


Fig. 3: The steps of the allocation process using our approach

In general, using a high-level layer containing simplified multi-variant components to abstract the details of the alternative, eases the information load of the allocation process. Instead of considering each component properties and other information (e.g., the communication links between connected components) from each alternative, the allocator considers only the abstracted variant properties, where the rest of the information is hidden away. Also, the concern of distributing connected CPU-GPU components to the same computation node is now implicit considered in the allocation process.

4 Running example

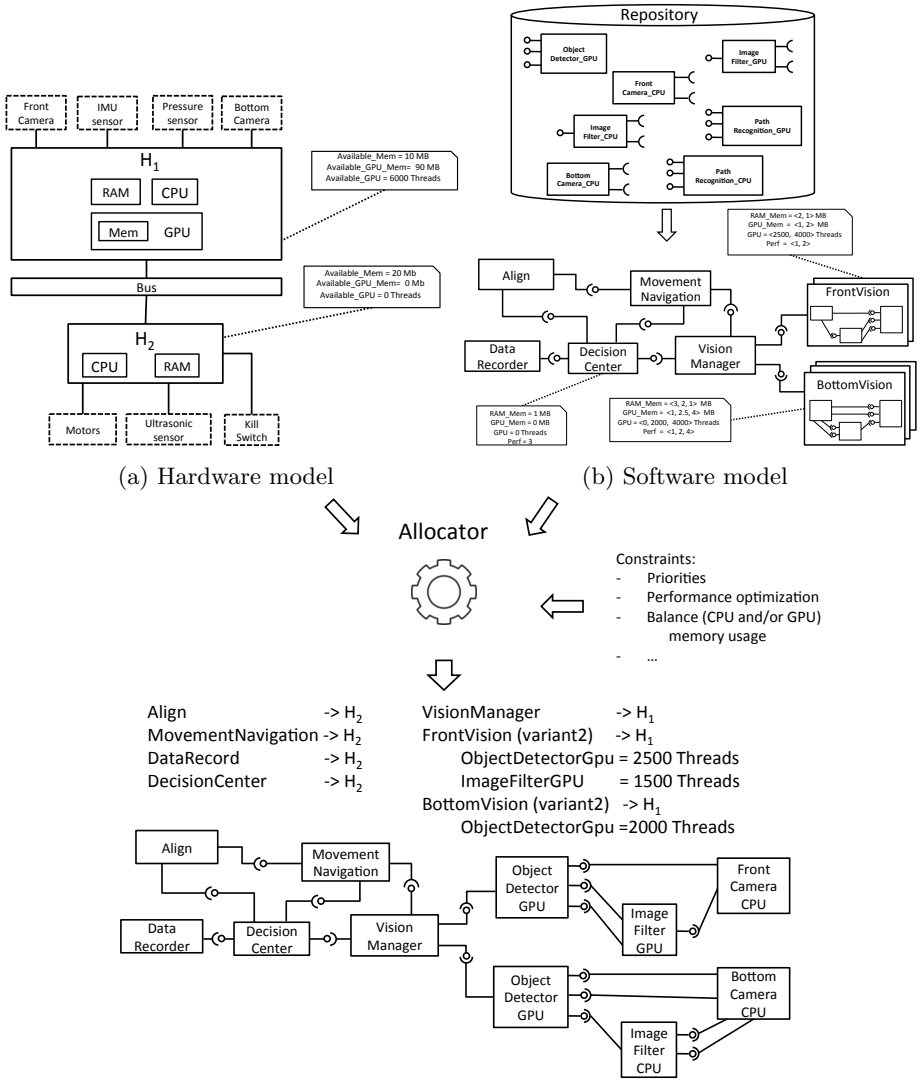
To illustrate the usage of our solution, we describe the allocation process of the underwater robot system, used during the paper. For the front and bottom vision systems, we developed several GPU and CPU-based components as follows. The repository of the vision systems contains three CPU-based components, i.e., *FrontCamera_CPU*, *BottomCamera_CPU* and *ImageFilter_CPU*, and two GPU-based component, i.e., *ImageFilter_GPU* and *ObjectDetector_GPU*. In our simple example, we characterized each CPU-based component with RAM memory usage and performance. The GPU-base components are defined by similar properties, such as the GPU memory usage and performance (e.g., execution time on GPU.

Dealing with a robot where performance and real-time responses may be crucial, a performance-driven component-based design of the front or bottom vision system would be composed mostly of GPU-based components. Not knowing beforehand various criteria such as the available hardware resources, the software-to-hardware allocation may be infeasible. For example, an alternative of the front vision system composed of only GPU-based components may demand more GPU computation threads than the hardware limitations, without even considering the resource requirements of the bottom vision system.

In the upper part of Figure 4, the hardware and the software architectures of the robot are presented. Figure 4a describes the hardware platform that contains two processing nodes, where only one has GPU capabilities. The nodes communicate over a CAN bus. To simplify our example, each hardware element is characterized by a minimal set of properties such as the available RAM and GPU memory and the GPU processing power. We use GPU threads capacity as a simple metric for specification of GPU computation power. In a more complex example, this property may be extended to include other metrics such as registers per thread.

Figure 4b describes the component-based design of the robot. The main component, *DecisionCenter*, controls the system settings (e.g., water pressure, color calibration specifications) and the robot missions. The robot propellers are controlled by the *MovementNavigation* component that maneuver the underwater robot based on the commands received from the *DecisionCenter* component. There are two vision systems, one for the front camera and another for the bottom camera. The *VisionManager* component takes decisions based on the information received from the vision systems. Both of the vision systems are constructed using components from the repository content. The front vision system is displayed as a component with two variants and the bottom vision system is described by a three variant component.

Each software component, such as *DecisionCenter* component, is characterized by extra-functional properties and performance, as seen in the Figure 4b. For the multi-variant components, we specify their properties in the following way. The *CPU_mem*, *GPU_mem* and *Perf* attributes are described as a sequence of values, where each value represents the resource usage of the corresponding variant. The *Perf* property is a sequence of variant values; the higher the value is, the



(a) Hardware model (b) Software model

(c) The allocation result and the system design

Fig. 4: The allocation process of the underwater robot demonstrator

better the variant performance is. For example, Table 1 presents the properties of the front vision multi-variant component. The component has two variants, where the first one is using 3Mb of RAM memory, 1 Mb of GPU memory and 2500 of GPU threads. The second variant, having two GPU-based components, uses more of the GPU resources (e.g., 2 Mb of GPU memory and 4000 Threads) than the previous variant.

Table 1: The properties of the 2-variant front vision component

Component variant	RAM_Mem (Mb)	GPU_Mem (Mb)	GPU (Threads)
1	3	1	2500
2	1	2	4000

Table 2 presents a detailed description of the properties of the two alternatives which corresponds to the front vision bi-variant component. Each alternative is composed of three components as follows. The first alternative has two CPU-based components and one GPU-based component, while the second alternative has two GPU-based components and one CPU-based component. The amount of information is much larger; each alternative is characterized by nine properties while a component variant has three properties.

Table 2: The properties of front vision alternatives

Alternative	Component	RAM_Mem (Mb)	GPU_Mem (Mb)	GPU (Threads)
1	FrontCameraCPU	2	0	0
	ImageFilterCPU	1	0	0
	ObjectDetectorGPU	0	1	2500
2	FrontCameraCPU	1	0	0
	ImageFilterGPU	0	1	1500
	ObjectDetectorGPU	0	1	2500

For our case study, we use simple properties such as static memory or thread usage; for synthesizing the variant properties, we use a simple addition operation. In general, other properties of a variant may also be derived from the components' properties of the corresponding alternative using different methods or techniques, when feasible.

The robot relies on the front vision system as the main vision system, and uses the bottom vision system as a secondary vision system when e.g., the front vision does not detect anything. Hence, the front vision has higher priority in accessing the GPU hardware resources. After the allocator receives the hardware and software information and other constraints such as priorities of the vision systems, the system allocation scheme is computed, as presented in the upper part of the Figure 4c. On the heterogeneous node H_1, both of the vision systems are allocated, while the rest of the components are allocated on the H_2 computing node. Having a higher priority, front vision system is allowed to access more of the GPU resources than the bottom vision system. Hence, the allocator selects the front vision variant that contains two GPU-based components: *ImageFilterGPU* and *ObjectDetectorGPU*. Because most of the GPU processing resources are

occupied by the front vision system, the allocator selects the bottom vision variant that contains only one GPU-based component.

In our simplified example, we used a small number of simple component properties. We did not consider other information such as the communication link properties between components. Also, the analysis and synthesizes of different properties into a set of variant properties is beyond the subject of this paper.

5 Related Work

There is a lot of research done in the allocation and software-to-hardware optimization domain, described in surveys like [3] and [4]. Several works present the tasks distribution used in the automotive industry [10], where an optimization method allocates task onto ECUs with different memory capacity and processing power. Various criteria are addressed by the allocation process, such as balancing local memory [12] for safety-critical multi-core systems or balancing the CPU processing power [8].

An allocator that covers CPU-GPU component allocation is described in our previous work [6]. The work translates an allocation optimization model into a mixed-integer programming solver (SCIP [1]). The solver, based on the software and hardware inputs, calculates feasible allocation schemes. The allocation model is formally describing the software component-model, the CPU-GPU hardware model and various allocation constraints such as balancing resources (i.e., memory, CPU and GPU computation power) and performance optimization. The work is limited by not considering multiple alternatives in the allocation process. The evaluation section describes the time and scalability limitations due to the amount of components and hardware information. Extending the work with our solution may result in a more accurate allocation with a better scalability property.

6 Conclusion

In this paper, we have proposed a solution to ease the software-to-hardware allocation process. Using a 2-layer component-based design for heterogeneous CPU-GPU embedded systems, we decreased the information load delivered to the allocator which can positively influence the total allocation process efficiency. Another advantage of our solution is that it can improve the allocation scalability by reducing the information load, making possible the allocator to handle more complex systems and component combinations. The disadvantage of our work is that it introduces new steps in the allocation process but can be worth accepting when the allocator gains benefits in efficiency.

For future activities, we propose to extend our previous work to include the solution presented in this paper and compare the efficiency results between the two work versions. Covering a limited number of simple component properties, a future work extension may include more extra-function properties and GPU specific properties to describe the software model. Another future work continuation

may include an adaptation or development of an existing method or technique, to allow us to synthesize the component properties to the variant properties.

Acknowledgment

Our research is supported by the RALF3 project - (IIS11- 0060) [4] through the 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. G. Campeanu, J. Carlson, and S. Sentilles. Component Allocation Optimization for Heterogeneous CPU-GPU Embedded Systems. In *The 40th Euromicro Conference on Software Engineering and Advanced Applications SEAA 2014*, 2014.
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. 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.
9. P. Michel et al. 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.
10. 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.
11. H. Nguyen. *GPU Gems 3.* Addison-Wesley Professional, first edition, 2007.
12. 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.
13. F. Xu and K. Mueller. Real-time 3d computed tomographic reconstruction using commodity graphics hardware. *Physics in medicine and biology*, 52(12):3405, 2007.

[4] http://www.mrtc.mdh.se/projects/ralf3/