Run-Time Component Allocation in CPU-GPU Embedded Systems

Gabriel Campeanu
Mälardalen Real-Time Research Center
Mälardalen University, Sweden
gabriel.campeanu@mdh.se

Mehrdad Saadatmand
Swedish Institute of Computer Science (SICS),
SICS Swedish ICT Västerås AB, Sweden
mehrdad@sics.se

ABSTRACT

Nowadays, many of the modern embedded applications such as vehicles and robots, interact with the environment and receive huge amount of data through various sensors such as cameras and radars. The challenge of processing large amount of data, within an acceptable performance, is solved by employing embedded systems that incorporate complementary attributes of CPUs and Graphics Processing Units (GPUs), i.e., sequential and parallel execution models.

Component-based development (CBD) is a software engineering methodology that augments the applications development through reuse of software blocks known as components. In developing a CPU-GPU embedded application using CBD, allocation of components to different processing units of the platform is an important activity which can affect the overall performance of the system. In this context, there is also often the need to support and achieve run-time component allocation due to various factors and situations that can happen during system execution, such as switching off parts of the system for energy saving. In this paper, we provide a solution that dynamically allocates components using various system information such as the available resources (e.g., available GPU memory) and the software behavior (e.g., in terms of GPU memory usage). The novelty of our work is a formal allocation model that considers GPU system characteristics computed on-the-fly through software monitoring solutions. For the presentation and validation of our solution, we utilize an existing underwater robot demonstrator.

CCS Concepts

•Computer systems organization → Embedded systems; •Software and its engineering → Allocation / deallocation strategies; *Software system structures*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SAC 2017,April 03 - 07, 2017, Marrakech, Morocco

Copyright 2017 ACM 978-1-4503-4486-9/17/04...$15.00

http://dx.doi.org/10.1145/3019612.3019785

Keywords

embedded systems, component-based development, GPU, CPU-GPU, component allocation, dynamic allocation, GPU monitoring, monitor

1. INTRODUCTION

Embedded systems are computation systems with limited resources and dedicated functionality. Nowadays, we have embedded systems in almost all human-used devices, from small size systems such as watches or telephones to large and complex systems such as spaceships or satellites. Many of the modern systems handle large amount of data in their functionalities. In addition, these systems may have real-time requirements, i.e., computing the outcome within a certain period of time. For example, the autonomous Google Self-Driving car [8] needs to process on-the-fly data from various sensors such as cameras, lasers and radars, in a specific time interval in order to adapt the car speed according to the detected traffic and street objects.

Traditional embedded systems confront new challenges when employed in the development of modern applications. One of the challenges is in handling great amount of data with an appropriate performance level. The CPU-based systems are known to process large amount of data in an inefficient way due to the CPU sequential architecture.

One solution to rapidly process large amount of data comes from employing GPUs. Having a parallel processing architecture, the GPU is known to rapidly handle large amount of data through data-parallel computations. Thus, it provides an improved performance compared to CPU, w.r.t. extensive data processing. One of GPU characteristics is that it can not function without a CPU. The recent technology breakthroughs permit development of embedded boards that contains CPUs and GPUs, such as NVIDIA [17] and UNIBAP e2050 [4].

Another trend in embedded systems is the employment of component-based development (CBD). As a software engineering methodology, CBD handles the increased complexity of embedded systems through composition of independent software units known as software components. The methodology has been successfully adopted by industry through dedicated component models such as AUTOSAR [18], Rubus [10] and Koala [20].

A system may greatly benefit from running the appropriate

part of the software onto the right hardware. The software-to-hardware allocation, known to be an NP problem [2], is an acknowledge challenge in designing an embedded system [21]. The component allocation in CPU-GPU embedded systems is not an easy task. On one hand we have two types of components characterized by different extra-functional properties as follows: *i)* components that use only the CPU characterized by typical system properties such as memory and CPU workload usage; and *ii)* components that utilize both the CPU and GPU are characterized, besides the typical properties, by GPU specific properties such as GPU memory usage and GPU workload usage. On the other hand, the hardware platform is composed of two different computation nodes (i.e., CPU and GPU) that have different properties such as available memory and GPU workload. A higher number of platform properties increases the allocation complexity.

In this paper, we develop a software allocation model that allocates components during run-time. The hardware targeted by our solution is CPU-GPU-based platforms. The allocator handles components that either utilize the CPU or have also GPU capabilities expressed using platform-independent properties such as GPU workload usage and GPU memory usage. The allocation model is formally described using a mathematical model that incorporates the definitions of the input software (i.e., component requirements) and hardware (i.e., available resources) models. The mathematical model is implemented using an integer programming model [7] that computes allocation schemes. Regarding the hardware, we equip each computation node with a monitoring solution to provide, during run-time, information on the resources availability status. We use a running underwater robot example during the paper to present the problem description and the solution overview; the same demonstrator is used to validate the feasibility of our proposed solution.

The remainder of the paper is divided as follows. Our work motivation is covered by Section 2 followed by the context background (Section 3). Problem description is introduced in Section 4, while Section 5 gives an overview of our solution. The realization of the allocation mathematical model is covered by Section 6. The solution is implemented as a MINPL model (Section 6) and validated using an underwater robot example (Section 8). The paper, after presents the related work in Section 9, concludes with a discussion about the limitations and future work directions of our work.

2. MOTIVATION

Nowadays, there exists a multitude of embedded systems of different sizes and complexities integrated in the daily human life. The small and simple embedded systems, characterized by a reduced system complexity, allow a tight off-line and static analysis of the temporal behavior using various methods such as fixed priority analysis (FPA).

The temporal behavior of large and complex embedded systems that may have real-time demands, is not always analyzable using off-line and static methods. The assumptions made by e.g., FPA may be violated by factors that appear during run-time. For example, due to the ever evolving aspects of the environment, the data load received by a system may greatly fluctuate from one moment to another. The transient data load is an aspect that may not be captured or predicted by an off-line and static analysis. One example that consistently deals with transient data load is the telecommunication networks. Inside these networks, big data loads arrive in an unpredictable and bursty manner that can not be anticipated and analyzed by static methods. Moreover, these networks can contain hundreds of computation nodes (e.g. Radio Network Controllers) that are distributed over large geographical regions. Being large systems, different extra-functional properties (EFPs) need to be considered in the software-to-hardware allocation, such as power consumption and heat generation. For example, to reduce the power consumption, some computation nodes may be switched off. Therefore, there are various factors such as the dynamic nature of data loads and optimizing system EFPs, that may uniformly require change in the allocation of software over the computation nodes of an embedded system.

3. CPU-GPU EMBEDDED SYSTEMS

When GPUs appeared in late 90s, they were used only for graphic-based applications. With the technology improvements, GPUs got increased computation power and also became programmable units. Having now easier means to work with GPUs, developers managed to port many non-graphical applications onto GPUs such as cryptography solutions [16].

Due to their unique (parallel) architecture characteristics, GPUs were successfully integrated into CPU-GPU embedded systems such as NVIDIA Jetson TK1 [17] and UNIBAP e2050 [4]. A CPU-GPU embedded system gets the most benefits by combining the two processing units. While the CPU is efficient in addressing sequential-based operations, the GPU proved its effectiveness in parallel processing operations. Equipped with hundreds of processing cores, the GPU manages to address data-parallel operations through the thousands of its computation threads. Figure 1 presents a high level architecture of an underwater robot that has three embedded boards from which two contain GPUs. A characteristic of the GPUs is that each unit has its own memory system. The embedded boards communicate using a communication buss.

4. PROBLEM DESCRIPTION

The run-time software-to-hardware allocation may be triggered by various factors (e.g., to save energy). The allocation needs to consider run-time software and hardware characteristics in order to compute feasible allocation schemes. There are several challenges in the allocation process as follows. The software needs to specify CPU-based attributes (e.g., memory usage) and GPU aspects (e.g., GPU memory usage). Having a higher number of properties increases the complexity of the allocation process. The hardware contains CPUs and GPUs and the information regarding the available resources needs to be determined during run-time.

To describe the problem in more details, we use a running example. The example is an underwater robot that contains three embedded boards connected by a CAN-bus link as described in Figure 1. The boards are connected to several sen-

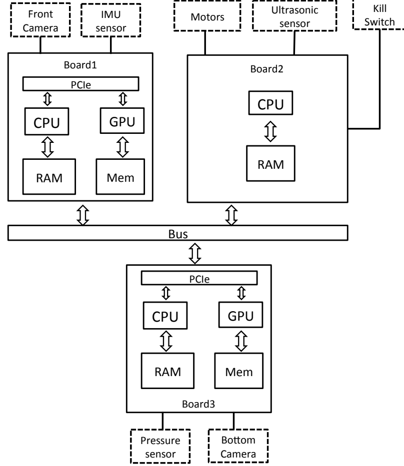

Figure 1: High-level architecture of an underwater robot that contains CPU-GPU embedded boards

sors and actuators such as pressure sensors, motors and cameras. The robot, using two cameras (front and bottom side) that provide a continuous stream of images, autonomously navigates under water to accomplish various missions such as detecting different objects (e.g., red buoys). A high-level component-based software architecture of the vision system is described in Figure 2. *DecisionCenter* is the component that, based on received data (e.g., water pressure), sets the configuration parameters (e.g., dive) and mission to follow. *Align* component aligns the robot with a certain object based on the feedback provided by *VisionManager* that processes images received from the robot's cameras.

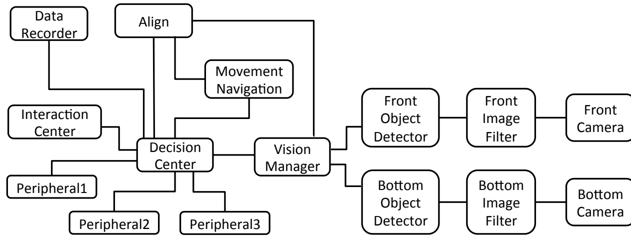

Figure 2: High-level component-based architecture of the underwater robot

The robot uses two (front and bottom) cameras for its vision system. Each camera is connected to a CPU-GPU embedded board as illustrated by Figure 1. Initially, the vision system, due to the GPUs easiness to process images, is distributed over *Board 1* and *Board 3* as follows. The part that handles the images received from front camera is allocated onto *Board 1*, and the part that handles bottom camera feedback is allocated onto *Board 3*. In order to save energy (in the detriment of performance), the robot switches to a SaveEnergy mode that shuts down one of the CPU-GPU boards (e.g. *Board 3*), reallocating all the components onto the rest of the system. The reallocation needs to be done during run-time. In the case when the robot requires a higher performance (e.g., many detected objects), it may switch to a Performance mode where the switched-off CPU-GPU board is again activated. When switching between modes, the components needs to be again on-the-fly reallocated.

5. SOLUTION OVERVIEW

For this initial stage of our solution, we characterize the software and hardware models with basic and abstract characteristics as follows. Each software components, either it has GPU capabilities or not, is characterized by four properties: *i)* the static (RAM) memory usage, *ii)* the CPU usage, *iii)* the GPU memory usage, and *iv)* the GPU usage. A component with GPU capabilities specifies (not null) values for all properties while a regular (CPU-based) component has null values for the GPU-related properties.

In the same manner, we characterize the computation nodes with four properties: *i)* the available static (RAM) memory, *ii)* the available CPU workload, *iii)* the available GPU memory, and *iv)* the available GPU workload.

For the *GPU usage* and *available GPU workload* properties, in this initial stage of our work, we use as a metric the number of threads. There are other parameters (e.g., registers per thread) that should be considered when describing the usage of GPUs, but we abstract them away and consider it as future work. For the *CPU usage* and *available CPU workload* properties, we use a reference unit (i.e., 1 CPU usage unit) and relate the properties to it. The properties related to memory are expressed in kB.

Due to the nature of the GPU, i.e., being always connected to a CPU that triggers all of the GPU activities, we see a CPU-GPU processing combination as a single processing node with (not null) value specifications of all four properties. The common CPU node defines null values for the two GPU related properties.

The information from the software and hardware models is fed to our proposed allocator solution that dynamically assigns components over hardware. The allocator is employed when e.g., a part of the system needs to be reallocated to save energy.

Figure 3 presents the overview picture of our solution. Whenever a system change occurs, the allocator is triggered. For example, in order to save energy, *Board 3* is switched off; thus, all components from *Board 3* need to be reallocated. The allocator is aware of the software model (i.e., the constraints of each component), hardware model (i.e., the availability status of the hardware recourses) and the current allocation scheme. Using this information, it computes an allocation scheme as described in the lower part of the figure. The allocation scheme presents on which node the components need to be allocated and how much of the resources they access.

Each processing node is equipped with a software monitoring solution that computes in real-time the available resources and saves the information to e.g., a log data-base file. In order to frequently refresh information on the available resources, the monitors query the hardware at a specific period of time that can be preset by the developer.

An alternative solution that tackles the overhead produced

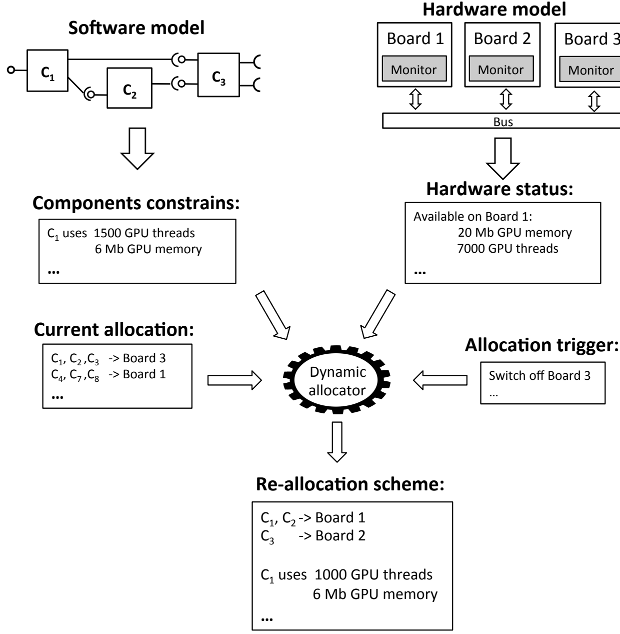

Figure 3: Overview of the dynamic allocation process

by periodically executing all the system monitors, is for the allocator to activate the monitors whenever it needs. The disadvantage of this solution is that an overhead is also introduced when the monitors are specifically triggering. In this work we assume that each node has a monitor solution that is periodically executed and stores the information into a log file.

6. ALLOCATOR REALIZATION

The formal model that captures the system characteristics and describes the allocation definition, is defined as follows:

6.1 Hardware and software model definitions

1. In our vision, the hardware system is a set H of k hardware computation nodes described by five functions: $Mem_Free : H \to \mathbb{R}_{\geq 0}$, $Cpu_Free : H \to \mathbb{R}_{\geq 0}$, $Mem_Gpu_Free : H \to \mathbb{R}_{\geq 0}$, $Gpu_Free : H \to \mathbb{N}_{\geq 0}$, where:

$$\forall h \in H, \text{then}$$
$$\begin{aligned} Mem_Free(h) &= \text{available static memory} \\ Cpu_Free(h) &= \text{available CPU capacity} \\ Mem_Gpu_Free(h) &= \text{available GPU static memory} \\ Gpu_Free(h) &= \text{available GPU capacity} \end{aligned}$$

2. We see the component-based application as set C of n components characterized by four functions: $Mem_Req : C \to \mathbb{R}_{\geq 0}$, $Cpu_Req : C \to \mathbb{R}_{\geq 0}$, $Mem_Gpu_Req : C \to \mathbb{R}_{\geq 0}$, and $Gpu_Req : C \to \mathbb{N}_{\geq 0}$, where:

$$\forall c \in C, \text{then}$$
$$\begin{aligned} Mem_Req(c) &= \text{static memory required by c} \\ Cpu_Req(c) &= \text{CPU workload required by c} \\ Mem_Gpu_R(c) &= \text{GPU static memory required by c} \\ Gpu_Req(c) &= \text{GPU workload required by c} \end{aligned}$$

6.2 Allocation definition

The allocator has the goal to dynamically map the component to nodes using the function $alloc : C \to H$ (i.e., one component goes to one hardware node), and to optimize the distribute of GPU threads among components through the function $gpu_alloc : C \to \mathbb{N}$, such that it satisfy the following constraints:

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

   $$\forall h \in H,$$
   $$\textstyle\sum_{c \in \{c | c \in C \wedge alloc(c) = h\}} Mem_Req(c) \leq Mem_Free(h)$$

2. The summed required CPU load of components placed on a node should not exceed the available node CPU workload.

   $$\forall h \in H,$$
   $$\textstyle\sum_{c \in \{c | c \in C \wedge alloc(c) = h\}} Cpu_Req(c) \leq Cpu_Free(h)$$

3. The GPU memory allocation is realized in two cases:

   - when h supports parallel execution, we need to make sure that the demands of all components that may run in parallel will not exceed the node resources. Hence, the summed required GPU memory of components allocated onto a GPU unit should not exceed the available GPU memory:

     $$\forall h \in H \text{ and}$$
     $$\forall c \in C \wedge alloc(c) = h \wedge gpu_alloc(c) > 0,$$
     $$\textstyle\sum_c Mem_Gpu_Req(c) \leq Mem_Gpu_Free(h)$$

   - when h does not support parallel execution, i.e., components are sequentially executed onto GPU, each component demands should not exceed the node resources. Therefore, the required GPU memory of each component allocated onto a GPU node should not exceed the available GPU memory:

     $$\forall h \in H \text{ and}$$
     $$\forall c \in C \wedge alloc(c) = h \wedge gpu_alloc(c) > 0,$$
     $$Mem_Gpu_Req(c) \leq Mem_Gpu_Free(h)$$

4. the distribution of the GPU computation threads among components is done in two cases:

- if h supports parallel execution, the threads allocation has a flexible manner as follows. A component is not restricted to use a fixed number of GPU threads for its execution. Instead, it can use fewer computation threads (with a different performance). For example, to filter an image that contains 4096 pixels, we assume that a component has a good performance when using 4096 threads (i.e., one thread per pixel). However, it can also use 1024 threads to process the same image by employing a single thread to process four pixels, but the execution time will increase. This way of utilizing less threads than requested, increases the possibilities of the allocator to compute feasible schemes.

  We start by calculating the total amount of computation threads demanded by components with GPU capabilities, that require to be allocated:

  $\forall h \in H$ and

  $\forall c \in C \wedge alloc(c) = h \wedge gpu_alloc(c) > 0,$

  $Total_Gpu_Req(h) = \sum_c Gpu_Req(c))$

  If there are more resources than requested, then each component receives as many threads it required. In the opposite case, the allocator distributes threads proportional with the demands of each component. For example, if a GPU has 4000 threads available and two components are to be allocated on it and demand 3000 threads and 2000 threads respectively, then the allocator provides 2400 and respectively 1600 threads to the components, using the following equations:

  $$\text{if } Total_Gpu_Req(h) > Gpu_Free(h):$$

  $$Gpu_Req(c) = \frac{Gpu_Req(c)}{Total_Gpu_Req(h)} * Gpu_Free(h)$$

- if h does not support parallel execution (i.e., sequential execution), each component receives the requested thread demands. In the case that the available thread resources are lower than the component requirement, then the component receives as much as is available.

7. IMPLEMENTATION

This section describes how we implemented the GPU monitoring system and the allocator.

7.1 The GPU monitoring system

The novelty of our work being related to GPUs, we present a solution that implements GPU monitoring systems. Our evaluation hardware contains an NVIDIA GPU hardware. Therefore, in this initial development stage of our allocator, we implemented a basic monitoring solution by using an existing library (i.e., *nvidia-smi*). The library displays, among other information, the following GPU details relevant to our work:

- the architecture employed by the GPU that reflects if the parallel execution feature is supported or not;
- the available GPU static memory; and
- the GPU utilization.

The monitor regularly examines the hardware at a particular period of time defined by the developer, e.g., every 3 seconds, and saves the information into a log file.

7.2 The allocator

As described in Section 6, the allocation model is composed of:

- two objective functions, i.e., one that maps components to nodes and the other that distributes computation threads among components; and
- several constraints, such as memory restrictions (see subsection 6.2).

The allocation model can be seen as a mixed-integer nonlinear programming (MINLP) model, where we optimize (i.e., maximize) the function that distributes computation threads to components subject to a number of constrains (i.e., inequalities). For solving the proposed model, we employ a mixed-integer solver (i.e., SCIP [1]) to computes solutions, given a specific input (hardware and software) model.

In order for the SCIP solver to interpret the allocation model, we translated the mathematical formulation into a mathematical program by using the ZIMPL language [12]. In Listing 1, we exemplify through a code line snippet, the allocation model translation. In the first two lines, the C and H are defined as the sets of components and nodes. The array variable *alloc*, defined as the Cartesian product of C and H, specifies the mapping between components and nodes using (0 and 1) boolean values. Lines 4 and 5 enforce that one component is allocated only to one node. The following lines (i.e., 6 and 7) implement the memory constraint (subsection 6.2, constraint 1), where *mem_host* is a function that previously defines the memory values of the nodes. In a similarly way, the *gpu_alloc* function is defined and maximized the GPU thread distribution.

Listing 1: Translation of the allocation model

```
1 set C :={"c1","c2","c3","c4","c5","c6","c7","c8"};
2 set H :={"h1","h2","h3"};
3 var alloc[CH] binary;
4 subto assign: forall <c> in C do
5  sum <h> in H : alloc[c,h] == 1;
6 subto constraint: forall <h> in H  do
7  (sum<c>in C:mem_comp[c]*alloc[c,h])<=mem_host[h];
```

8. EVALUATION

This section contains an evaluation of the feasibility of our method by applying it on the underwater robot case study. Table 1 presents the components allocation, as follows. Each component of the robot solution, illustrated on the hand left

Table 1: Component allocation schemes for the underwater robot

Component	Allocated on board (before re-allocation)	GPU usage (threads)	Allocated on board (after re-allocation)	GPU usage (threads)
Align	3	0	2	0
BottomCamera	3	0	1	0
BottomImageFilter	3	3000	1	1000
BottomObjectDetector	3	3000	1	1000
FrontCamera	1	0	1	0
FrontImageFilter	1	3000	1	3000
FrontObjectDetector	1	3000	1	3000
DataRecorder	2	0	2	0
DecisionCenter	2	0	2	0
InteractionCenter	2	0	2	0
MovementNavigation	3	0	2	0
Peripheral 1	1	0	1	0
Peripheral 2	2	0	2	0
Peripheral 3	3	0	1	0
VisionManager	3	0	1	0

side of the table, is initially allocated on a particular embedded board and has a particular GPU usage expressed in the number of used threads. We mention that each GPU contains a total of 8000 threads and allows parallel activities execution. The components that execute on *Board 3*, after it is turned off, are redistributed by the allocator as presented in the right hand side of the table with their new GPU usage properties. Components *BottomImageFilter* and *BottomObjectDetector* that require GPU, are allowed each to access 1000 threads (see last equation, subsection 6.2). Also, they are permitted to be parallel executed with the rest of the components with GPU capabilities, that reside on *Board 1*.

Although the underwater robot demonstrator exists and its software application follows the described component-based architecture (see Figure 2), our run-time allocator solution is not integrated into the system. Further evaluations need to look into how the solver affects the robot resources and the overhead introduced when computing re-allocation schemes and moving components between hardwares.

9. RELATED WORK

The software-to-hardware allocation concern is discused in a large body-of-knowledge. Much of the work is done on task allocation onto CPU-based systems. Static solutions are covered by different surveys [13] and methods for dynamic task allocation are examined by various studies such as [19].

Regarding heterogeneous embedded systems, there are several works addressing the allocation issue. We mention the work of R. Li et al. [14] that proposed an automated allocation process in which it considered different metrics such as processor utilization and data flow latency.

It is worth to mention a static component optimization allocator for CPU-GPU embedded platforms which is introduced by Campeanu et al. [5]. Similar to our approach, the authors use a formal description of the allocation model and employ the same SCIP solver [1] to compute allocation schemes. There are several differences compared to our solution, as follows. The allocator proposed by Campeanu is more complex than our solution, by considering the component communication aspects which provides a more accurate allocation schemes. It also considers different optimization criteria such as balancing system memory usage and optimizing the GPU performance. The drawback of the static allocator is that it always considers a parallel execution of the components with GPU capabilities, regardless of the hardware capabilities. Our work is more precise by checking against the hardware capabilities (i.e., if it supports or not parallel execution) and adjusting the allocation schemes accordingly.

Regarding monitors, D. Haban et al. [9] introduced the software monitoring solutions to aid scheduling tasks with random execution times in real-time systems. The authors present the performance degradation of (CPU-based) systems with less than 0.1% when using software monitoring. For our work, we implemented the GPU monitoring as a task executed on CPU that may affect the CPU workload. There are different works that implement GPU monitors for different purposes. For example, data flows monitoring solution are implemented to enable packet-level simulations for large systems [3]. Another example employs GPU monitors to dynamically balance GPU bandwidth use [11].

10. DISCUSSION AND LIMITATIONS

This paper provides a solution to dynamically allocate components over CPU-GPU platforms. The hardware resource availability information is provided in real-time to the allocator by monitoring software solutions. In addition to the hardware information, the allocator uses details on the software model (e.g. components properties such as memory usage) and the ongoing allocation scheme. The novelty of our solution is a formal allocation methods that considers GPU characteristics. Basically, our solution computes on-the-fly allocation schemes and distributes the available GPU resources (e.g., GPU threads) to components.

There are several limitations of our solution, and due to the work being focused on GPUs, we discuss only the constraints related to the GPU part of our solution. To the best of our knowledge, there is no developed component model mechanism to allow parallel execution of components onto

GPU. Previous work that has been done that introduces the GPU-aware component notion and enriches component models with GPU specific artifacts (e.g., GPU ports), considers sequential component execution onto GPU [6]. There are other limitation regarding the parallel execution of components with GPU capabilities. A more accurate analysis should examine how many components are able to be executed in parallel. For example, having three components with GPU capabilities as illustrated in Figure 4 that require to be allocated, the allocation model should consider only the parallel execution of components C_2 and C_3, while C_1 is sequentially executed.

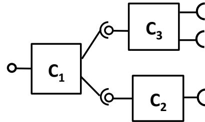

Figure 4: Three connected components to be allocated

Another limitation is that the allocation process considers only a few basic GPU-aware component properties such as GPU global memory usage and GPU load usage (expressed in GPU threads usage). Other properties that have an important role in computing a more precise allocation scheme, were abstracted away from this initial solution. For example, we did not consider GPU properties such as shared memory usage and registers per GPU thread usage. The described limitations are seen as future work directions to extend and improve our work.

Acknowledgments

Our research is supported by the RALF3 project [15] - (IIS11-0060) through the Swedish Foundation for Strategic Research (SSF).

11. REFERENCES

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

[2] S. K. Baruah. Task partitioning upon heterogeneous multiprocessor platforms. In *IEEE Real-Time and Embedded Technology and Applications Symposium*. Citeseer, 2004.

[3] 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.

[4] F. Bruhn, K. Brunberg, J. Hines, L. Asplund, and M. Norgren. Introducing radiation tolerant heterogeneous computers for small satellites. In *2015 IEEE Aerospace Conference*. IEEE, 2015.

[5] G. Campeanu, J. Carlson, and S. Sentilles. Component allocation optimization for heterogeneous CPU-GPU embedded systems. In *40th EUROMICRO Conference on Software Engineering and Advanced Applications*. IEEE, 2014.

[6] 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.

[7] R. S. Garfinkel and G. L. Nemhauser. *Integer programming*, volume 4. Wiley New York, 1972.

[8] Google. Google Self-Driving Car Project. https://www.google.com/selfdrivingcar/. Accessed: 2016-08-18.

[9] D. Haban and K. G. Shin. Application of real-time monitoring to scheduling tasks with random execution times. *IEEE Transactions on software engineering*, 16, 1990.

[10] 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.

[11] 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.

[12] T. Koch. *Rapid Mathematical Prototyping*. PhD thesis, Technische Universität Berlin, 2004.

[13] Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv.*, 1999.

[14] R. Li, R. Etemaadi, M. T. Emmerich, and M. R. Chaudron. An evolutionary multiobjective optimization approach to component-based software architecture design. In *2011 IEEE Congress of Evolutionary Computation (CEC)*. IEEE, 2011.

[15] Malardalen University. RALF3 - Software for Embedded High Performance Architectures. http://www.mrtc.mdh.se/projects/ralf3/. accessed September 28, 2016.

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

[17] NVIDIA. NVIDIA Jetson TK1. http://www.nvidia.com/object/jetson-tk1-embedded-dev-kit.html, accessed September 14, 2016.

[18] A. D. Partnership. Technical overview v4.2. http://www.autosar.org, (accessed September 28, 2016).

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

[20] R. Van Ommering, F. Van Der Linden, J. Kramer, and J. Magee. The Koala component model for consumer electronics software. *Computer*, 33, 2000.

[21] T.-Y. Yen and W. Wolf. *Hardware-software co-synthesis of distributed embedded systems*. Springer Science & Business Media, 2013.