Partitioning Algorithm for Mixed Criticality Systems *

Emilio Salazar and Alejandro Alonso

Universidad Politécnica de Madrid
{esalazar, aalonso}@dit.upm.es

Abstract

Mixed criticality systems are composed of a set of applications with different criticality levels. The interest relies on the possibility of running these applications on a single processor, with advantages on cost, size, weight and energy consumption. The use of partitioning for developing these kinds of systems is a suitable approach. A related issue is how to partition a system. This paper describes an automatic partitioning algorithm. It relies on restrictions to state requirements that the system partitioning must meet. They provide great flexibility for meeting requirements derived from the user or from specific functional and non-functional requirements

1 Introduction

Mixed criticality systems are composed of a set of applications with different criticality level. Current processors power allows for integrating them in a single computer. This approach reduces costs, volume, weight, and power consumption. Certification procedures for safety-critical systems require the system to be certified as a whole. This implies the certification of non-critical applications in a mixed criticality system, which is very costly. Moreover, if any of these components are modified, the new system has to be certified again. System partitioning is a way of dealing with these issues. A hypervisor provides temporally and spatially isolated partitions. Applications with different criticality levels are allocated on different partitions. In this way, it is possible to certify partitions in an independent way.

The development of partitioned systems requires additional development activities, such as system partitioning. Its purpose is to define a set of partitions or virtual machines, which are characterised by the core where they are executed,

*This work has been partially funded by the Spanish Government, project HI-PARTES (TIN2011-28567-C03-01), and by the European Commission FP7 programme project Multi-PARTES (IST 287702).

the assigned resources (memory, CPU share, etc.), its operating system, and the allocated applications. This process is done by a system integrator engineer. There is a lack of tools to assist in this process, which makes it error-prone, due to the amount of parameters and design considerations that are involved.

MultiPARTES [1] is a FP7 project aimed at developing tools and solutions for building trusted embedded systems with mixed criticality components on multicore platforms. The approach is based on an innovative open-source multicore platform virtualization layer based on the XtratuM hypervisor [2]. A software development methodology and its associated toolset [3] [4] is being developed in order to enable trusted real-time embedded systems to be built as partitioned applications, in a timely and cost-effective way.

In this toolset, the generation of the partitioning is done in three stages: generation of the system partitions, allocation of resources to partitions, and validation. The motivation for this approach is to ease the integration of support for additional non-functional requirements. This paper describes an algorithm for automatically generating a system partitioning. It takes as inputs the system models, that include applications, hardware, operating systems, and the hypervisor. It relies on restrictions, which defines requirements that the resulting partitioning must meet. They allow a great level of flexibility to the tool. Restrictions can be defined by the system integrator or automatically generated from applications requirements. There is no need to modify the partitioning for considering the restrictions.

The structure of this paper includes a review of relevant related works, the description of the underlying system model and the partitioning constraints. Then, it is described an automatic partitioning algorithm. Finally, an use case illustrates the behaviour of the algorithm.

2 Related work

This paper focuses on the generation of system partitions. This stage takes the system model and the partitioning restrictions as input, and as a result, it generates the allocation of the applications to the partitions. It is important to note that in this stage the partitions are created but not scheduled.

There is a large effort of research on scheduling partitioned systems (e.g. [5, 6, 7, 8, 9, 10, 11, 12, 13] However, in all of this research the starting point is a system where the applications are already allocated to their partitions. The aim of this paper is to automatize the step before the scheduling, i.e. the process of creating and allocating applications to partitions. The toolset also schedules the resulting partitioning schema, although this stage is out of the scope of this paper.

The allocation of applications to partitions can be reduced to a resource allocation problem which is commonly addressed with graph coloring techniques. The coloring of a graph is a well-known problem and there is a large amount of work done [14, 15, 16].

Often, resources are associated with the colors used for coloring graphs.

Therefore, minimizing the number of colors is minimizing the number of resources used. Nonetheless, finding the smallest number of colors that are needed to color all of the vertices of a graph is a known NP-Complete problem [17] and for this reason greedy algorithms have to be used.

The algorithm that is described in this paper is based on a greedy algorithm used in the compiler register allocation, which can be also reduced to a resources allocation problem. A lot of research has been done on the register allocation problem [18, 19, 20, 21, 22]. The proposed partitioning algorithm adapts the algorithm proposed by [20] and then improved by [22] by exchanging registers with partitions and temporal variables with applications.

As stated before, the partitioning schema and the execution plan are generated in different stages. This approach makes it possible to deal with these problems (i.e the partitioning and the scheduling) with more specific and optimized algorithms. Additionally, this approach keeps the scheduling and the partitioning algorithms isolated, which provides an easier way of improving each of them independently. Another important advantage of this approach is the integration of new non-functional properties. This approach has been successfully proven in the integration of the MultiPARTES' tools.

The opposite approach is taken by [23], who proposes a Tabu Search-based algorithm that, at the time, creates a partitioning schema where the development costs are minimized and the tasks are schedulable. Based on the criticality of each task, the algorithm proposed by [23] determines whether a task can or cannot be allocated to the same partition as another task. It provides a separation graph for capturing additional separation requirements. By means of this graph, it makes it possible to define which tasks are not allowed to share the same partition.

However, the approach suggested by [23] omits other parameters that may force two tasks to be allocated on different partitions (e.g. operating system, core affinity, processor family, etc.). It also lacks support for partitioning restrictions; for example, it cannot force two tasks to share the same partition, nor can it pre-allocate tasks to predefined partitions.

3 Problem statement

System Model

In the context of this paper, the system is composed of a set of applications that run on an execution platform. An application is considered to be a software entity that provides a closed set of functionalities. It can interact with other applications to perform its duties. The application is defined by a model, that allows adding annotations for describing non-functional requirements.

The execution platform is composed of the hardware platform, a hypervisor, and the set of operating systems that can be run on top of the hypervisor. The hardware platform comprises all computational devices needed by the final system. The hypervisor provides a set of partitions or virtual machines, where

applications are run in isolation. A partition is characterized by the assigned resources, the operating system and a set of applications. The resources assigned should be sufficient for running the applications with the expected performance and meeting time requirements.

Partitioning restrictions

Partitioning restrictions are a set of statements that defines requirements to be met by the system partitioning. As such, the corresponding algorithm must consider them in order to generate a valid partitioning. The sources of restrictions are:

- *Implicit*: The implicit restrictions are related to information that application models must include for any system, such as the criticality level, resource needs, operating system, and processor type. The toolset automatically analyzes these properties and extracts a set of restrictions. For instance, it is not possible to allocate two partitions with different criticality level or requiring different operating system in the same partition.
- *Explicit*: This type of restrictions is the basis for considering additional non-functional requirements (NFR) in the partitioning. If the toolset has to support one additional NFR, the first step is to provide means for annotating this information in the application model. Then, it is possible to use a transformation to generate automatically partitioning restrictions. For instance, if the toolset is going to support security requirements, it is needed to ensure that applications with sensitive information are not allocated in the same partition.
- *System integrator*: He can provide additional restrictions based on his experience or on particular requirements of a given system.

The analysis of the types of constrains required for a number of systems, has concluded with the identification of a minimum set of general statements. They are sufficient for stating application concerns with respect to partitioning:

- *Application a must go with applications* $v = \{b_1, b_2..., b_n\}$. This restriction forces to allocate application a into the same partition as applications in the list v. Let $\gamma(a) = \{b_1, b_2..., b_n\}$ denote the function that, given an application, returns the set of applications v that must execute in the same partition than a.
- *Application a must not go with applications* $v = \{b_1, b_2..., b_n\}$. This restriction forces to allocate application a into a different partition than the applications from list v. Let $\delta(a) = \{b_1, b_2..., b_n\}$ denote the function that given an application, returns the set of applications v that must execute in a different partition than a.

– *Application a must be allocated to partition p.* This restriction forces to allocate the application a into the partition named p. Let $\xi(a) = p$ denote the function that, given the application a, returns the partition p where it must be allocated.

– *Application a must not be allocated to partition p.* This restriction forces to allocate the application a into a different partition than p. Let $\phi(a) = p$ denote the function that, given the application a, returns the partition p where a must not be allocated.

It is important to note that the source of the restrictions is not meaningful for the partitioning algorithm. They are used for constructing the initial graph. Then, this graph is the only information used by the algorithm.

Problem statement

The aim of the partitioning algorithm is to generate an allocation of applications into partitions, which is valid if it meets the following criteria:

– All applications must be allocated to partitions

– All partitioning restrictions are met.

It is difficult to define what is an optimal solution to this problem. From the point of view of graph theory, it is when the minimum number of colors is used. In an embedded system, there are other factors to consider, such as time behavior, power consumption or performance. The approach in this work is to generate an initial solution meeting the previous criteria, and to generate alternative solutions if needed. The toolset components that allocate resources or validate the partitioning may request additional valid allocations, in order to decide which is the best one in a particular system.

4 Partitioning Algorithm

The proposed algorithm is based on colored graphs. Colored graphs are a special type of graphs in which the vertices are labeled (or colored) according to certain restrictions. When the colored elements are the vertices, it is called vertex coloring. The most typical vertex coloring is called proper vertex coloring. In a proper vertex-colored graph no two adjacent vertices share the same color.

More precisely, the algorithm is based on an important property of a colored graph: the chromatic number. The chromatic number is the smallest number of colors that are needed to (proper) color all of the vertices of a graph. Finding the chromatic number is a known NP-Complete problem [17]. For this reason, a modified version of the greedy algorithm proposed by [20] and later improved by [22] is used.

A system is defined as:

– A set of applications $\{a_1, a_2..., a_n\}$.

– A set of partitioning restrictions $\{\omega_1, \omega_2..., \omega_n\}$.

Let G be a colored graph unequivocally defined by the tuple (V, E, C, L, M, N):

– V is the set of vertices.

– E is the set of edges (u, v) where u is linked to v. Given that the graph is not directed, the order of the vertices denoting an edge is not significant, $(u, v) = (v, u)$.

– C is the set of colors.

– L is the set of allocations where an allocation (u, c) means that the vertex u is colored with the color c.

– M is the set of forbidden allocations where an allocation (u, c) means that the vertex u is cannot be colored with the color c.

– N is the set of vertices that have not been colored yet.

In addition, the following functions are defined:

– Let $adjacents(u) = \{v_1, v_2...v_n\}$ denote a function that returns the set of adjacent vertices of the vertex u.

– Let $colors(u) = \{c_1, c_2...c_n\}$ denote a function that returns the set of colors of the vertex u.

– Let $adjacentsColors(u) = \{c_1, c_2...c_n\}$ denote a function that returns the set of colors used by the $adjacents(u)$.

– Let $forbiddenColors(u) = \{c_1, c_2...c_n\}$ denote a function that returns the set of colors that cannot be used to color u.

– Let $score(u, c) = s_{uc}$ denote a function that scores the coloring of the vertex u with the color c. The higher is the score, the more desirable is the coloring of u with c.

4.1 Building the Graph

The graph construction process can be broken down into the following stages:

– *Vertices population.* The graph is initialized with a vertex for each application of the system.

– *Vertices merging.* Given a restriction *application a must go with set of applications* v, a merge of a with each of the applications of v is carried out. Each merge operation is basically:

  – Creation of a new vertex v_{ab}.
  – Elimination from the graph of the two merged vertices v_a and v_b.

  - Merging in v_{ab} all the colors defined for v_a and v_b.
  - Merging in v_{ab} all the forbidden colors defined for v_a and v_b.

- *Adjacent vertices.* For each restriction *application a must not go with applications of the set v*, all applications of v are defined as adjacent vertices of a.

- *Pre-coloring vertices.* For each restriction *application a must be allocated to partition p*, the vertex created for a is pre-colored with the color created for p:

  - A new color c_p is created for p. The same partition always gets the same color. Therefore, if other application was already allocated on partition p, the color retrieved for p must be the same c_p.
  - The vertex v_a is colored with c_p

- *Forbidding colors.* For each restriction *application a must not be allocated to partition p*, the color created for p, c_p, is forbidden to the application a:

  - A new color c_p is created for p. The same partition always gets the same color. Therefore, if other application was already allocated on partition p, the color retrieved for p must be the same c_p.
  - TThe color c_p is added to the forbidden color list of v_a.

4.2 Coloring the Graph

This subsection summarises the graph coloring process, which is splited in two main steps. The first step is in charge of simplifying the graph and creating a queue with all non-colored vertices, sorted by their degree. A second step pops the vertices from the queue and assigns them a color.

4.2.1 Simplifying the graph

The aim of this step is to reduce the graph by removing its vertices in inverse order of degree and pushing them into a queue. The simplification of the graph is actually carried out on a copy of the G, G', in order to avoid losing information. Only non-colored vertices are extracted from the graph. Therefore, those vertices that were pre-colored remain in the graph. The process is as follows:

- The lowest degree vertex v_n of G is selected.
- v_n is removed from G.
- v_n is pushed in Q.
- If G contains non-colored vertices go to step 1. Otherwise, start the vertices coloring.

4.2.2 Coloring Vertices

This step can be broken down into the following stages:

- *Retrieving candidate colors of the vertex v.* For each vertex v extracted from the queue Q, a set of candidate colors is computed. To begin with, all colors already created in the graph, C, are valid candidates. However, colors used by the live adjacent vertices of v and colors forbidden for v must be removed. If the resulting set is empty, a new color is created.

- *Coloring the vertex v.* Once the candidate color set is computed, only one of these colors can be used to color v. For this purpose, the function $score(u, c)$ is defined. This function returns a score that indicates how desirable the coloring of v is with a specific candidate color. The color that receives the highest score is the one used to color v.

- *Multiple candidate color.* Only one color is used to color a vertex when multiple candidate colors are available. However, all of the candidate colors are valid colors. This means that the provided solution is only one of the possible colorings of the graph. When the first vertex v_{first} with multiple candidate colors is found, the graph is saved before coloring v_{first}. Then, the vertex is colored with the highest-scored color, c_{first}, which is added to the v_{first}'s forbidden colors list, as this color cannot be used again in further solutions.

- *Alternative colorings.* When an alternative coloring is requested, the graph is restored to the same state as in v_{first}, the first vertex with multiple candidate colors is found. However, when v_{first} must be colored, the color c_{first} (used in the first coloring) is now in the forbidden color list of v_{first}. Therefore, c_{first} is discarded from the valid candidate colors of v_{first}. As a result, v_{first} is colored with the next highest-scored color. This process ends when all nodes are colored.

5 Use Case

Three use cases in the MultiPARTES projects (wind power, aerospace, and video surveillance) have relied on this partitioning algorithm for generating their system partitioning. In this subsection, a more complex case is used for illustrating the algorithm behaviour. The system is composed of a set of applications $\{a, b, c, d, e, f, g, h, i\}$. Application models include information, such as their criticality level and operating system, as shown in table 1.

Application	a	b	c	d	e	f	g	h	i
Criticality Level	A	C	C	C	A	A	A	A	A
Operating System	ORK	Linux	Linux	Linux	XAL	XAL	XAL	ORK	XAL

Table 1: Applications characteristics

The initial phase of the algorithm automatically generates a set of implicit restrictions, for ensuring that applications with different operating system or with different criticality level are not allocated in the same partition. The initial graph takes into account these restrictions.

In this use case, the application models also include annotations about security requirements, which may imply their allocation to different partitions, in order to prevent information leakage between them. A transformation could generate the related explicit restrictions from the application models, which are the following:
$\{\delta(b) = d, \delta(f) = i\}$

Finally, the system integrator can add restrictions for enforcing a given allocation for particular applications, in order to meet requirements based on their experience or on certification standards. In this case, the restrictions below are supposed to be defined in this way:
$\{\gamma(f) = e, \xi(f) = 1, \xi(g) = 1, \xi(h) = 2\}$

The first step of the algorithm is building the graph:

- *Vertex creation.* Since there are nine applications, it is necessary to create 9 vertices.
- *Pre-coloring vertices.* $\xi(f) = 1, \xi(g) = 1, \xi(h) = 2$ state that applications f and g must be allocated to partition 1, whereas application h must be allocated to partition 2.
- *Vertex merging.* $\gamma(f) = i$ implies that applications f and i must be allocated to the same partition, so vertices v_f and v_i must be merged into a new vertex v_{fi}. When two vertices are merged, the new vertex inherits all of the pre-colors that the original vertices had. In this case, v_{fi} inherits the pre-coloring 1 from v_f.
- *Adjacent vertices.* $\delta(b) = d$ implies that application b and d must not be allocated to the same partition. $\delta(f) = e$ states that applications f and e must be allocated to different partitions.

Figure 1 is the resulting graph from the first step. Dotted lines represent γ constraints. Bold black lines show δ explicit constraints, and regular black lines show implicit δ constraints. In rest of figures, the type of restriction, and hence the type of arch, is not meaningful for the algorithm. Vertex are labelled with a triplet: vertex identifier, associated applications, and a number that represents a color.

Once the graph is built, the second step is the graph coloring. This step is, in turn, broken down into two stages:

- *Simplify the graph.* In this stage, a queue is made by removing the lowest degree non-colored vertex from the graph, as shown in figure 2.
- *Vertices' coloring* (see figure 3). Vertices are extracted from the queue in LIFO order and then colored. When no colors are available, a new

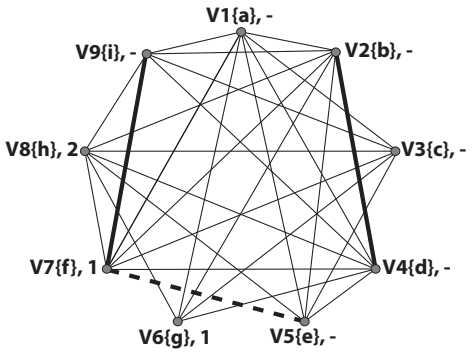

Figure 1: Initial system

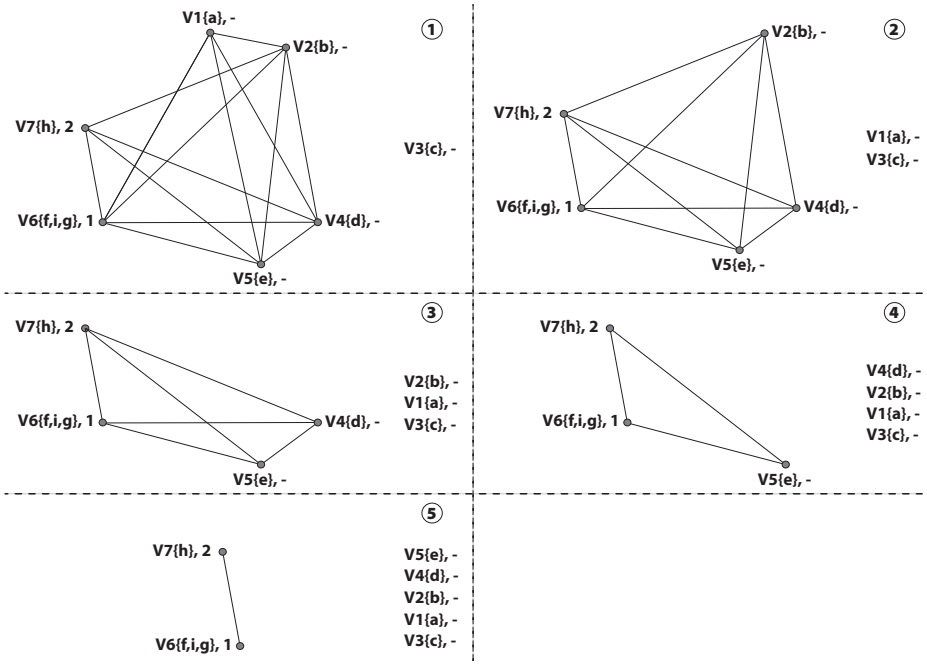

Figure 2: Simplify the graph

color is created (see figure 3.1, 3.2 and 3.3). It is possible that a vertex has multiple valid colors available (see figure 3.5). In order to choose the color, a scoring function evaluates all of the possible allocations. The allocation with the highest score is chosen. In the case of Figure 3.5, vertex v_c is colored with color 4, which is added to the forbidden colors list of vertex v_c. If an alternative solution is requested, v_c would be colored with color 5.

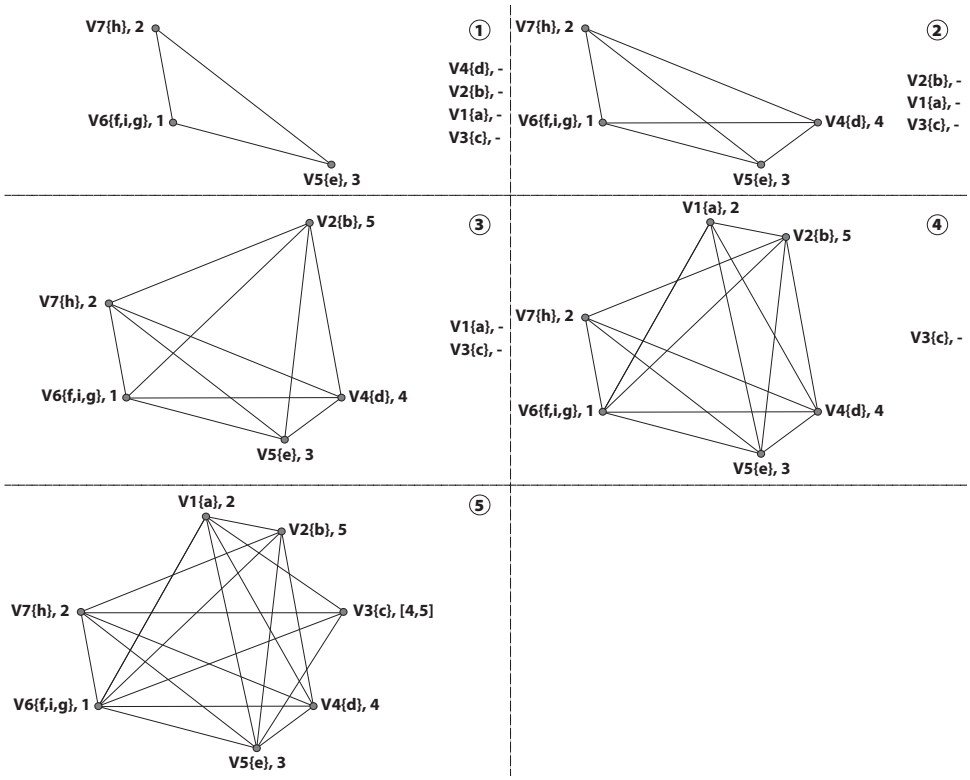

Figure 3: Vertices' coloring

As a result of the graph coloring, the initial partitioning schema is:
$\{f, i, g\}, \{a, h\}, \{e\}, \{d, c\}, \{b\}$

If an alternative schema is requested, the algorithm provides: As a result of the graph coloring, the initial partitioning schema is:
$\{f, i, g\}, \{a, h\}, \{e\}, \{b, c\}, \{d\}$

In both schemata, all of the applications with different criticality levels are allocated to a different partition. Applications f and i are both allocated to the same partition($\gamma(f) = i$). Since it was requested f to be allocated to partition 1 ($\xi(f) = 1$), i is allocated to this partition, as well. Also, g has to be allocated to partition 1 ($\xi(g) = 1$). Applications b and d cannot be allocated to the same partition ($\delta(b) = d$).

6 Conclusions

This paper describes an algorithm for generating a system partitioning in a mixed criticality embedded system. The inputs are the application models and a set of restrictions, that define requirements on the partitioning. The aim is to

provide an allocation of the applications to partitions, in such a way that the set of restrictions is met. The use of restrictions is an important contribution of the paper. The aim is to ensure that partitioning requirements can be modified, without the need to change the partitioning algorithm. This is a basic mean for easing the extension of the support to non-functional requirements by the MultiPARTES toolset, where restrictions are automatically generated from their specification. This toolset includes additional tools for assigning computational resources to the partitions and for validating the resulting system design.

A set of complex scenarios has been used for validating the outcomes of the algorithm. In addition, it has been served to generate the partitioning on three industrial use cases. The results so far have been successful. The formal demonstration of the algorithm is under development. An algorithm for assigning resources to partitions and for generating an scheduling plan is currently under test.

References

[1] MultiPARTES: Multi-cores Partitioning for Trusted Embedded Systems, Available: www.multipartes.eu

[2] Masmano M., Ripoll I., Crespo A., Peiro S.: XtratuM for LEON3: an Open-Source Hypervisor for High-Integrity Systems. Embedded Real Time Software and Systems (ERTS2 2010), May 2010.

[3] A. Alonso, E. Salazar, and M.A. de Miguel, A Toolset for the Development of Mixed-Criticality Partitioned Systems, 2nd Workshop on High-performance and Real-time Embedded Systems (HiRES 2014) Vienna, Austria

[4] Salazar E., Alonso A., de Miguel M.A., de la Puente, J.A. "A Model-Based Framework for Developing Real-Time Safety Ada Systems". In H.B. Keller, et al (eds.), Reliable Software Technologies — Ada-Europe, LNCS 7896, pp. 126–141. Springer-Verlag, 2013.

[5] Brocal, Vicent, et al. "Xoncrete: a scheduling tool for partitioned real-time systems." Embedded Real-Time Software and Systems (2010).

[6] Biondi, Alessandro, G. Buttazzo, and Marko Bertogna. "Schedulability analysis of hierarchical real-time systems under shared resources." RETIS Lab, Scuola Superiore Sant'Anna, Italy, Technical Report TR-13-01 (2013).

[7] Easwaran, Arvind, et al. "A compositional scheduling framework for digital avionics systems." Embedded and Real-Time Computing Systems and Applications, 2009. RTCSA'09. 15th IEEE International Conference on. IEEE, 2009.

[8] Easwaran, Arvind, Madhukar Anand, and Insup Lee. "Compositional analysis framework using EDP resource models." Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International. IEEE, 2007.

[9] Lackorzynski, Adam, et al. "Flattening hierarchical scheduling." Proceedings of the tenth ACM international conference on Embedded software. ACM, 2012.

[10] Kuo, Tei-Wei, and Ching-Hui Li. "A fixed-priority-driven open environment for real-time applications." Real-Time Systems Symposium, 1999. Proceedings. The 20th IEEE. IEEE, 1999.

[11] Feng, Xiang, and Aloysius K. Mok. "A model of hierarchical real-time virtual resources." Real-Time Systems Symposium, 2002. RTSS 2002. 23rd IEEE. IEEE, 2002.

[12] Deng, Zhong, J. W. S. Liu, and J. Sun. "A scheme for scheduling hard real-time applications in open system environment." Real-Time Systems, 1997. Proceedings., Ninth Euromicro Workshop on. IEEE, 1997.

[13] Jin, Hyun-Wook, and Sanghyun Han. Temporal partitioning for mixed-criticality systems. Emerging Technologies and Factory Automation (ETFA), 2011 IEEE 16th Conference on. IEEE, 2011.

[14] Brélaz, Daniel. "New methods to color the vertices of a graph." Communications of the ACM 22.4 (1979): 251-256.

[15] Johnson, David S. "Worst case behavior of graph coloring algorithms." Proc. 5th SE Conf. on Combinatorics, Graph Theory and Computing. 1974.

[16] Maffray, Frédéric. "On the coloration of perfect graphs." Recent Advances in Algorithms and Combinatorics. Springer New York, 2003. 65-84.

[17] Garey and Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness. 1979. ISBN: 0-7167-1044-7.

[18] Appel, Andrew W., and Lal George. "Optimal spilling for CISC machines with few registers." ACM SIGPLAN Notices. Vol. 36. No. 5. ACM, 2001.

[19] Chaitin, Gregory J., et al. "Register allocation via coloring." Computer languages 6.1 (1981): 47-57.

[20] Chaitin, Gregory J. "Register allocation and spilling via graph coloring." ACM Sigplan Notices. Vol. 17. No. 6. ACM, 1982.

[21] Briggs, Preston, Keith D. Cooper, and Linda Torczon. "Improvements to graph coloring register allocation." ACM Transactions on Programming Languages and Systems (TOPLAS) 16.3 (1994): 428-455.

[22] George L., Appel A.W.: Iterated register coalescing. TOPLAS 18(3), 300–324 (1996).

[23] Tamas-Selicean, Domitian, and Paul Pop. Design optimization of mixed-criticality real-time applications on cost-constrained partitioned architectures. Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd. IEEE, 2011.