Actas de las XXXVI Jornadas de Automática, 2 - 4 de septiembre de 2015. Bilbao
ISBN 978-84-15914-12-9

An Automatic System Partitioning Algorithm for Mixed-Criticality Systems

Emilio Salazar[1] and Alejandro Alonso[1]

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

Abstract

The continuous improvement of processors computational power and the requirements on additional functionality is changing the way embedded systems are built. Applications with different safety requirements are executed in the same processor giving rise to mixed-criticality systems. The use of partitioned systems is a way of preventing undesirable interferences among applications. Still, the development of partitioned systems requires additional development error-prone activities often done by the system integrator. This paper describes an algorithm aimed at supporting the development of partitioned systems by generating system partitioning automatically, taking into account applications and platform characteristics.

1 Introduction

Traditionally, applications with different safety requirements have been allocated to different computers. However, processors power allows current computers to integrate a large number of applications in a single multi-core computer. As a consequence, there is a significant reduction of costs, volume, weight, and power consumption. In addition, certification for safety-critical systems is a very expensive process that involves the whole system, regardless of the safety requirements of each application. Moreover, they must be reevaluated with each single change on any of the applications.

The use of virtual machines or partitions, provided by an hypervisor, is an approach aimed at reducing costs and making the certification procedures easier is the system partitioning. In a partitioned system, applications of different criticality levels run on different partitions. The hypervisor ensures temporal and spatial isolation between partitions. A faulty application does not interfere on the behavior of other applications. In this way, it should be possible to make an independent certification of applications.

The development of partitioned systems, however, requires additional activities, such as the partitioning of the system. It requires the system integrator to consider a large amount of features, related with applications and execution platform. On the other hand, partitioned systems are a fairly new approach and accordingly there is a lack of tools for supporting its development. As a consequence and despite of their complexity, often these error-prone activities are crafted by the system integrator.

In the framework of the MultiPARTES [1] project, a toolset aimed at supporting the development of partitioned mixed-criticality systems was developed. A key part of this toolset is the algorithm for automatically generating a system partitioning consistent with the input models.

This paper describes the proposed algorithm and the tests and analysis that have been made in order to validate it. In addition, it is detailed how the heuristic search that is done for searching for an optimal solution and the provided mechanisms to make the algorithm more general and flexible.

2 Related Work

There is an important amount of research projects aimed at partitioned mixed-criticality systems.

In the context of the ASSERT [7] project was produced a toolset [8] which has been extended for supporting partitioned systems [2]. This tool however requires the system partitioning to be provided in advance.

CERTAINTY [3] dealt with the certification process for mixed-critical embedded systems. It is proposed a unified semantics for systems and languages with mixed-criticality concern [9]. Main outcomes [11] were a scalable interference analysis framework, a scheduling policy for mixed-criticality multi-core system based on flexible time-triggering and a resource sharing and virtualization mechanism. Additionally, a WCET analysis tool was extended to include more architectures [10]. Nonetheless, partitions are crafted in advance.

RECOMP [4] studied how to enable cost-efficient certification and re-certification of safety-critical

ISBN 978-84-15914-12-9

systems and mixed-criticality systems. To that end, several validation, verification and timing analyses for component validation were chosen and extended [12, 13]. Moreover, several operating systems [14] for a number of hardware platforms [15] were extended. In addition, a set of tools were select to create the different tool-chains supporting the development and certification life-cycles [16]. Similarly with [2, 3], RECOMP assumes that the partitions are already defined.

In short, so far, research projects have been aimed at improving the support of the development of partitioned mixed-criticality systems. However, there still exist a lack of research in the automatic system partitioning generation.

EMC^2 [5] aims at finding solutions for dynamic adaptability in open mixed criticality real-time systems through the entire life-cycle. A toolset addressing the modeling and analyzing multi-domain mixed-criticality applications is to be created. This toolset currently supports schedulability analysis based on Response Time Analysis and C code generation. It will also support a complete implementation of both, single and multiprocessor platforms.

DREAMS [6] aims at virtualized mixed criticality systems on multicore platforms. It will deliver architectural concepts, meta-models, virtualization technologies, model-driven development methods, tools, adaptation strategies and validation, verification and certification methods.

Ongoing research projects such as EMC^2 and DREAMS assume also that the partitions are defined and fixed from the beginning. Of course, as ongoing projects, they have not produced yet all of their outcomes and accordingly, this assumption might change.

For this reason, one of the goals of the toolset developed in the MultiPARTES project was to automatically generate a ready-to-run partitioned system where the system partitioning has been deduced based on the input models provided by the engineer.

To the best of the authors knowledge, the closest approach regarding automatic system partitioning algorithms is published in [17]. In this paper, Tamas et al. proposed a Tabu Search based algorithm that creates a partitioning schema where the development costs are minimized and the tasks are schedulable.

Contrary to the Tamas' approach, the algorithm presented in this paper decouples the partitioning logic from the scheduling policy. As a consequence, changes on any of these critical processes do not imply to modify the algorithm which, in turn, decreases the cost of adapting the algorithm to new scenarios. Moreover, parameters such as operating system, core affinity, processor family, hardware requirements, etc are supported and properly managed to assure a consistent system partitioning. In addition, in this paper, a wider range partitioning requirements are supported.

3 System Modeling

In mixed-criticality systems, applications run on a particular execution platform regardless of the application criticality. An application is a software component that provides a well-defined functionality. The impact on the system mission of a fault in a specific functionality determines its criticality.

The execution platform is the environment where applications run. The most important actors of the execution platform are the hardware, the hypervisor and the operating system. Hardware are all the computational devices (e.g. processor, memory, I/O devices, etc.) where the system executes.

The hypervisor is a layer of low-level software that provides virtual machines (i.e. partitions) where a fixed set of applications run on a specific operating system and a known set of resources such as processor budget and memory. The hypervisor provides also temporal and spatial isolation. This crucial feature in mixed-criticality systems means that a fault or misbehavior on a partition does not impact on the behavior of the other partitions.

In addition to applications and execution platform, there may exist a number of non-functional requirements (NFR) that the final system must meet. NFR may have a sound impact on the final system behavior, correctness and efficiency. Relevant examples of the former requirements are real-time, safety or security.

Following the Model-Driven Architecture (MDA) [20], all these data is captured by means of models. The system is thus composed by a set of models which are based on specifically designed meta-models, for allowing a complete description of these entities.

System models are the input of the partitioning algorithm. These models hold the most relevant information to the partitioning algorithm:

- *Operating System.* This model holds partitioning relevant information about the operating system such as its version, library paths or the operating system processor family.

ISBN 978-84-15914-12-9

- *Hypervisor.* The goal of this model is to describe important information about the hypervisor which impacts on the partitioning algorithm. For instance the processor family or the library paths.
- *Hardware Platform.* This model describes the underlying hardware. Remarkable data held in this model is the amount of cores, memory and I/O devices.
- *Application.* In this model, application information related to the partitioning is contained. Relevant examples of these data are application real-time parameters (i.e. period, deadline and computation time), application criticality, application operating system, applications hardware requirements, etc. It is worth to mention that, in order to provide backward compatibility, there are two different application models:
  - *Basic application.* This model provides only the basic required data for the partitioning algorithm and it is intended to model legacy applications which have no available documentation.
  - *UML-MARTE application.* This models provides a highly detailed vision of the internal structure of the application which enables additional features such as automatic code generation or time analysis.
- *Partitioning constraints.* This model is intended to model the NFRs of the system which must be met in the final system partitioning. It is further discussed in section 5.1.

Further details about the modeling of the partitioning algorithm inputs can be found in [18, 19].

4 The Problem of Partitioning Mixed-Criticality Systems

The result of partitioning a mixed-criticality system is a system partitioning. A system partitioning is a particular allocation of the system applications to partitions. This allocation must however meet a number of conditions to be considered valid:

- All applications are allocated to, at least, one partition.
- All partitions host, at least, one application.
- Resources allocated to partitions do not exceed those available.
- All non-functional requirements are met.

Owning to the size and complexity of the partitioning problem, an approach based on dealing with the problem as a whole does not seem to be practical. Instead, a divide-and-conquer approach has been taken by breaking down the partitioning problem into three smaller problems:

- *Allocate applications to partitions.* The problem is how to group the initial set of applications into different partitions. The allocation must meet a number of partitioning requirements in order to assure a valid result.
- *Allocate partitions to processing resources.* Once the partitions are defined, they must be allocated to different processing resources. This problem is analogous to the bin packing problem which is, in turn, a well known NP-Hard problem.
- *Schedule partitions.* Partitions must be scheduled so that all applications meet their deadlines but taking into account that there are two scheduling levels. On one hand, each partition hosts a number of applications that are executed according to a local scheduling policy. On the other hand, each partition is scheduled based on a global scheduling policy. This problem is called hierarchical scheduling and it is a studied NP-Hard problem.

This paper is focused on the first of the above points. For this reason, hereinafter the *partitioning problem* shall refer to the problem of allocating applications to partitions. Accordingly, the *partitioning algorithm* shall refer to the algorithm created to handle the partitioning problem.

5 Modeling the Allocation of Applications to Partitions

The partitioning problem has been formally modeled in order to make the use of mathematical algorithms easier. This representation is also used as internal representation in the partitioning algorithm.

The choose of the mathematical representation is indeed of a critical importance as it is the base for the rest of the algorithm. For this reason, a study of the representation used in other fields for solving similar problems has been carried out. As a result of this study, the allocation of intermediate variables to machine registers has arose as a very close problem. This problem, denoted as the *register allocation problem*, has been and still

ISBN 978-84-15914-12-9

is profusely studied by the compiler research community [24, 25, 26, 27, 23, 31, 32, 33] as it is of a great practical importance.

The approach provided by [25] and then improved by [26, 27] is probably one of the most used approaches in modern compilers. The key idea behind this approach is to model the register allocation as graph (i.e interference graph) and then k-color this graph where K is the amount of available machine registers.

A *vertex colored graph* is a graph where vertices may be tagged (or colored) according to a given policy. One of the most usual coloring policies is the *proper vertex coloring*. In a proper vertex colored graph, no two adjacent vertices share the same color. A *k-coloring* of a graph is a coloring that uses, at most, k colors for proper coloring the graph.

The partitioning algorithm presented in this paper is based on the same idea: modeling the problem of allocating applications to partitions as a vertex colored graph which is built as follows:

- *Vertices.* Each vertex represents an application of the system.
- *Edges.* Each edge links two applications that cannot be allocated to the same partition.
- *Color.* Each color represents a partition.

5.1 Partitioning Constraints

Applications (i.e. vertices) and partitions (i.e. colors) are properly represented in the graph. However, the modeling of the non-functional requirements (NFR) needs further considerations.

There are a number of NFR that have an important relevance in the the allocation of applications to partitions. Some relevant examples are safety, real-time and security requirements. A common factor of these requirements are that they imply a set of restrictions on the allocation of applications to partitions. Two application with different safety criticality levels cannot be allocated in the same partition. The same holds for applications with sensible information.

The proposed approach is to create a set of simpler *partitioning constraints* that specify the effects of a NFR on the final allocation of applications to partitions. In other words, the idea is to design a set of simple constraints that makes it possible to generate automatically a set of partitioning constraints to ensure the fulfilment of specific NFR. This approach provides several advantages:

- *NFR agnostic partitioning algorithm.* Virtually any type of NFR can be processed with the same and unmodified partitioning algorithm as long as each NFR can be expressed in terms of partitioning constraints.
- *Composition of NFR.* The fulfilling of all the partitioning constraints implies fulfilling of all the NFR as well.
- *Makes the integration easier.* Individually, all of the partitioning constraints are intentionally very simple and in addition, the number of them is small. Therefore, the partitioning constraints can be used by different third-party tools as a common language to express impact of additional of NFR on the system partitioning.

5.1.1 Constraints Sources

- *Implicit constraints.* The resulting system works only if all of these constraints are met. For instance, two applications with the different operating systems cannot be allocated to the same partition as each partition hosts only a single operating system. Often, these requirements are automatically deduced from the data extracted of the input models.
- *Explicit constraints.* As a rule, contrarily to the explicit constraints, these constraints must be explicitly provided. Two main kinds of explicit constraints are:
  - *Non-functional constraints.* System properties such as real-time, safety or security are provided with these constraints. For this reason, they are crucial to make the final system work properly.
  - *System integrator constraints.* These constraints deliver the engineer background and expertise.

5.1.2 Constraints Types

- *Basic constraints.* After studying several use cases, the following constraints arose as the most important constraints. So far, with these constraints was possible to describe all of the needed requirements. In addition, these are the sole constraints that have a direct impact on the graph.
  - *Application A must not be allocated along with B.* This constraints forces the algorithm not to allocate A to the same partition as B. In terms of the graph, it is represented as an edge between vertices A and B.

ISBN 978-84-15914-12-9

  - *Application A must be allocated to partition P.* This constraint forces the algorithm to allocate the application A to the partition P. In the graph, it is represented by pre-coloring the vertex A with the color that stands for the partition P.
  - *Application A must be allocated together with B.* This constraint forces the algorithm to allocate applications A and B to same partition. This is translated to the graph by pre-coloring both applications with the same color.
  - *Application A must not be allocated to partition P.* This constraint indicates the algorithm to avoid allocating application A to the partition P. In the graph, the color P is added to the banned colors list of vertex A.

- *Combined constraints.* These convenient constraints are sets of basic constraints that represent a number of common partitioning requirements.
  - *Application A must be allocated to core C.*
  - *Application A must be allocated to processor P.*
  - *Application A must be allocated to a core of the processor family F.*
  - *Application A must be allocated at the address X.*
  - *Application A requires access to hardware H.*
  - *Application A must run on hypervisor H.*
  - *Application A requires partition system privileges.*
  - *Application A requires floating point support.*

6 The Partitioning Algorithm

The goal of the partitioning algorithm is to find a valid allocation of applications to partitions based on the provided mathematical model (i.e. graph) described in section 5.

Furthermore, there are several principles that has driven the design of the partitioning algorithm:

- *General.* It must be able of handling a wide range of systems from different domains. A way of achieving the desired levels of generality is by modeling the problem with a high level of abstraction. To that end, as stated in section 3, MDA proposes a methodology where models are the center of the developing process. According to the MDA strategy, relevant information is captured by a number of models while the algorithm itself is embedded in different model-to-model transformations.

- *Flexible* The algorithm must be able to model and process all of the NFR regardless of their origin. This is achieved by partitioning constraints which are described in subsection 5.1.

- *Adaptable.* The nature of the partitioned systems makes difficult to provide a general definition of what is a correct and optimal system partitioning. For this reason, the algorithm shall provide means for customizing both concepts according to the parameters defined in each system.

6.1 Correctness

As stated in section 4, there exist a minimum amount of necessary conditions that a system partitioning must meet to be considered valid.

The proposed approach to find a valid system partitioning is to *proper vertex color the graph built in section 5* . Provided that this graph was properly built, the resulting coloring is equivalent to a system partitioning where all of the aforesaid conditions are met:

- *All vertices are colored.* By definition, all of the vertices of a proper colored graph are colored with, at least, one color. From the point of view of the partitioning problem, this property ensures that all of the applications are allocated to, at least, one partition.

- *Each color is used in, at least, one vertex.* By definition, all colors created for proper coloring a colored graph have been used to color, at least, one vertex. Analogously, in the partitioning problem, this property ensures that there is no empty partitions or, in other words, it ensures that all partitions host, at least, one application.

- *There is no to adjacent vertices with the same color.* By definition, a proper colored graph cannot use the same color in two adjacent vertices. As a consequence, a proper colored graph is equivalent to a system partitioning that by definition meets all the partitioning constraints.

Furthermore, the solution is guaranteed as there always exist a (naive) proper coloring (i.e. each vertex is colored with a different color) provided that the graph does not have vertices connected directly back to themselves. The naive coloring

ISBN 978-84-15914-12-9

is usually not the desired result though as it is discussed in section 6.2.

6.1.1 Color Filters

Several NFR can be difficult to model by means of a colored graph. For instance, in the partitioning of a high-integrity system, it may be important to guarantee that no partition uses more than a given maximum amount of processor time. In such cases, the algorithm provides *color filters*.

A color filter is a function that rejects the use of a specific color for coloring a particular vertex according to an user-specific policy. Each vertex is colored only with colors that have validated all the filters. In addition, there is no limit in the amount of color filters so, a set of color filter may be provided in order to ensure the fulfillment of different NFR.

In the high-integrity system example, a color filter can be provided for avoiding the allocation of applications to partitions that have exceeded the maximum allowed processor time.

6.2 Optimality

A graph may have multiple proper colorings and consequently multiple valid system partitioning may be produced for a single system. For this reason it is also important to define criteria for selecting the optimal partitioning for a given system.

The notion of optimal system partitioning is not absolute. It depends on the specific requirements of a particular system. It some cases, it may be interesting to minimize the number of partitions. In other cases, the cores load balancing can be more relevant.

For this reason, the partitioning algorithm does not impose any particular optimal search function but rather it provides a customizable function.

If no specific optimal function is provided, the default optimal in the partitioning algorithm is the smallest amount of partitions (see subsection 6.2.1). This optimal makes sense in many partitioned systems as it is commonly desirable to retrieve the smallest, fastest and simplest system possible. Yet, as said, there are scenarios where the amount of partitions is not a priority (e.g. high-integrity systems) and where a different optimal search function may be provided.

6.2.1 Smallest Amount of Partitions

Generating a system partitioning with the smallest amount of partitions is equivalent to coloring a graph with the smallest amount of colors. However, coloring a graph with the minimal amount of colors is a known NP-Hard problem [34]. In other words, there is no known algorithm that provides optimal solutions in polynomial time.

Heuristic algorithms do not ensure an optimal solution but often run in polynomial time. For this reason, they are commonly used for attack NP problems. There is a vast amount of heuristics for coloring a graph with the smallest amount of colors, but as a rule, a coloring heuristic proceed as follows:

1. Sort vertices according to a given heuristic in a list of vertices.
2. Color the first vertex of the list.
3. Remove the colored vertex from the list.
4. If the list is empty, finish. Otherwise, if the heuristic is has dynamic vertex ordering go to 1. Otherwise, go to 2.

Despite the amount of heuristics, in practice [35], three heuristics are the most commonly used:

- *Largest-First (LF)* [36] is a static vertex ordering heuristic that sorts vertices in decreasing order of degree. LF starts to color with the most connected vertex (i.e. highest degree vertex) because this vertex is probably the most difficult vertex to color given that is the vertex with the highest amount of neighbors.
- *Largest-Saturation-First (LSF)* [37] is a dynamic vertex ordering heuristic that colors the vertex with the highest saturation. The saturation of a vertex is the number of different colors used by the vertex neighbors.
- *Smallest-Last (SL)* [38] is a static vertex ordering heuristic based on the same idea as LF. However, the vertices sorting process is refined which avoids certain faults of LF.

After studying a number of use cases, it was stated that the amount of vertices is very rarely *big* (i.e. more than 100 vertices). Therefore, given that the today's computing power makes insignificant the required time for coloring a graph of less than 100 vertices, the default behavior of the proposed algorithm is to color the graph using the three aforementioned heuristics and choose the best result.

ISBN 978-84-15914-12-9

6.3 Alternative System Partitioning

There is not an upper bound to the amount of colors of a vertex. This means that, regardless of the optimal search function and the amount of color filters, a vertex may be have multiple candidate colors when a graph is proper colored.

Each vertex is colored with a single color for each system partitioning that is generated. However, when a vertex has multiple candidate colors, the preferred color is selected and removed from the candidate colors list. The preferred color is the first color of the vertex candidate colors list. The default order is LIFO but if it is needed, the system engineer may provide a *specific vertex candidate colors sorting function*.

When an alternative system partitioning is requested, the graph is re-colored starting by the first vertex with multiple candidate colors that was found in the previous coloring. Then, the preferred color (i.e. the first color of the vertex candidate colors list) is chosen for coloring the vertex and then removed from the list of candidate colors of the vertex.

The subsequent vertices with multiple candidate colors are colored using their preferred colors, but unlike the first vertex, the preferred color is not removed from the list. The reason is that different colorings of the first vertex with multiple colors may or may not affect to the rest of vertices with multiple candidate colors as it might happen that the sole difference between two different system partitioning was the coloring of the first vertex with multiple candidate colors.

The resulting colored graph represents thus an *alternative system partitioning* which has also assured the correctness as it is a valid proper coloring of the initial graph. The level of optimality of the different alternative system partitioning depends on the provided optimal search function and candidate colors sorting function.

An alternative system partitioning may be very helpful in the development of complex system, where many analysis aimed at verifying the correctness of the final system are involved. This feature enables the system integrator to ask for an alternative system partitioning if the original system partitioning is rejected by later non-functional analysis such as schedulability or performance.

7 Algorithm Synthetic Tests

These tests have been developed with the following goals in mind:

- Determine the scalability of the algorithm with the default optimal search function.
- Determine the impact on the generated system partitioning of the color filters using the default optimal search function (i.e. least amount of partitions).
- Verify the correct implementation of the different procedures used in the algorithm.

7.1 Generation of a Test Case

A model representing the system to be partitioned (i.e. project model) is created with a random number of application models, hypervisor models, operating system models, hardware platform models (i.e. cores, processors and I/O devices) and explicit partitioning constraints.

In turn, each of these components is randomly created. For instance, each hardware platform models has a random number of processors that have a variable number of cores working at different frequencies.

Upper and lower bounds are defined to control the project model complexity. For example, if a complex project model is desired, a lower bound of 50 applications may be set. If it is required to generate a mono-core hardware platform, a upper bound for processors and cores my be defined.

7.2 Scalability

These tests are aimed at defining the average execution time of the algorithm in projects of different sizes and complexities.

Notwithstanding that the scalability is always a desirable feature, it actually is not a priority in the partitioning algorithm design since it is very unlikely that a system have more than 100 applications. In addition, even in case of hundreds of applications, the execution times of the algorithm are of a few seconds as tests have shown.

Figure 1 depicts how the amount of applications impacts on the global execution time of the algorithm.

In this particular case, the test case is composed of 200 randomly generated projects with sizes between 5 and 200 applications each of them. In addition, in order to stress the algorithm, a complex environment is generated: a hardware platform with 6 mono-core processors, four different operating systems and 2 hypervisors.

Such complex scenario ensures an important amount of implicit partitioning constraints which, at the end, leads to a complex graph to color. The

ISBN 978-84-15914-12-9

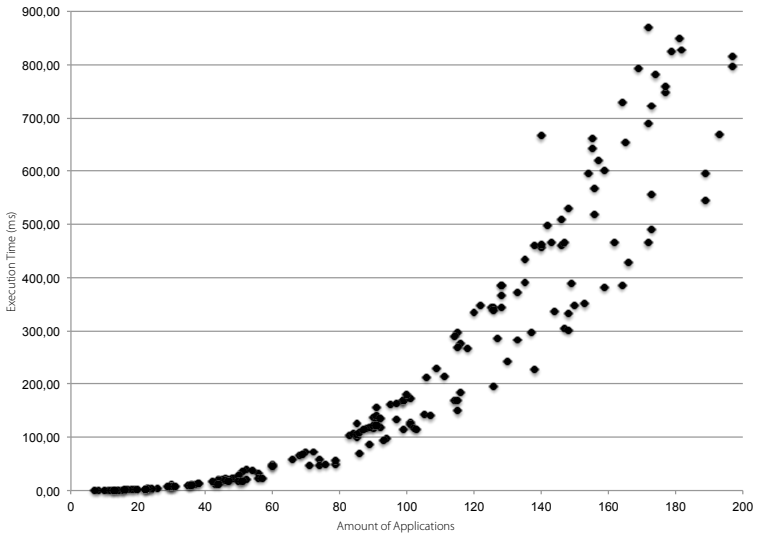

Figure 1: Execution time of projects with sizes between 5 and 200 applications

heuristic used for coloring the graph is LSF without any additional color filter.

Figure 1 shows the amount of vertices (i.e. applications) has an important impact on the execution time: while 100 vertices require less than 200 ms, the double of them require almost 900 ms.

However, figure 1 also states that even in case of huge projects with 200 applications, the execution times are below the second. One second is a very reasonable time for an algorithm that is aimed at improving the development process of partitioned systems, not to be executed on-line in the final partitioned system.

7.3 Color Filters

These tests are aimed at determining the impact of introducing color filters in the algorithm.

Some high-integrity systems must work even in overload situations. The approach commonly taken is to define a maximum processor time for each partition so that none of the partitions can use more than the defined processor time. This ensures that, even in case of overload, all partitions will have available time to produce their results.

This test case analyzes the impact of limiting the processor time available for each partition. The scenario and the used heuristic is the same as in subsection 7.2.

Figure 2 depicts clearly the impact of limiting the processor time on the generated partitions. Triangles are the amount of generated partitions when the processor load of each partition is limited to 5%. On the other hand, lines show the amount of generated partitions when the processor time is limited to 85%.

As expected, the lower is the available processor time, the higher is the amount of generated partitions. When partitions have up to 85% of processor time, the amount of partitions is under 15. However, if the amount of available time processor goes down to 5%, the amount of required partitions grows up to 30 partitions.

Referencias

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

[2] Delange, Julien, Christophe Honvault, and James Windsor. "Model-Based Engineering Approach for System Architecture Exploration."

[3] CERTAINTY (Certification of Real-Time Applications designed for mixed criticality). EU FP7 project ref. 288175. http://www.certainty-project.eu

[4] RECOMP (Reduced certification costs using trusted multi-core platforms). Artemis EU project ref. 100203. http://atcproyectos.ugr.es/recomp/

[5] EMC^2 (Embedded Multi-Core systems for Mixed Criticality applications in dynamic and changeable real-time environments). Artemis project ref. 621429. http://www.artemis-emc2.eu

[6] DREAMS (Distributed Real-time Architecture for Mixed Criticality Systems). EU FP7-

ISBN 978-84-15914-12-9

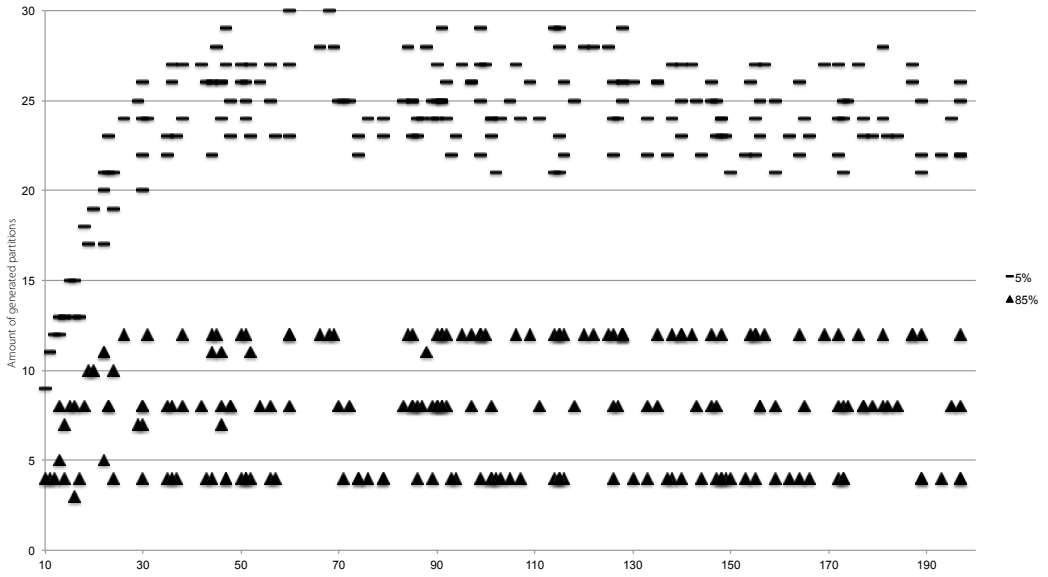

Figure 2: Amount of generated partitions when the processor time of each partition is bounded.

ICT project ref. 610640. http://www.dreams-project.eu

[7] ASSERT (Automated proof-based System and Software Engineering for Real-Time systems). EU FP-IST project ref. 004033. http://www.assert-project.net

[8] Perrotin, M., et al. "TASTE: a real-time software engineering tool-chain overview, status, and future." SDL 2011: Integrating System and Software Modeling. Springer Berlin Heidelberg, 2012. 26-37.

[9] D2.3 - Modelling Languages and Models. http://www.certainty-project.eu

[10] D3.3 - Identification of resources access and timing analysis principles. http://www.certainty-project.eu

[11] D5.1 - Interference Analysis and Isolation Mechanisms Report. http://www.certainty-project.eu

[12] D2.2.1 - Report on component validation methods and technique. http://atcproyectos.ugr.es/recomp

[13] D2.3.1 - Validation, verification and timing analyses. http://atcproyectos.ugr.es/recomp

[14] D3.3 - Operating system support: Architecture, Implementation, Evaluation. http://atcproyectos.ugr.es/recomp

[15] D3.4 - HW support for operating systems and applications: Concept, Implementation, Evaluation. http://atcproyectos.ugr.es/recomp

[16] D2.5 - Guidelines for developing certifiable systems and integration with existing tool flows. http://atcproyectos.ugr.es/recomp

[17] Tamas-Selicean, Domitian, and Paul Pop. Design optimization of mixed- criticality real-time applications on cost-constrained partitioned architectures

[18] Emilio Salazar, Alejandro Alonso,Jorge Garrido. Mixed-criticality design of a satellite software system. In E. Boje, X. Xia (eds.), Proceedings of the 19th IFAC World Congress, IFAC-PapersOnLine, 2014 ISBN 978-3-902823-62-5; DOI 10.3182/20140824-6-ZA-1003.02002.

[19] Alonso, Alejandro, and Emilio Salazar. "Toolset for Mixed-Criticality Partitioned Systems: Partitioning Algorithm and Extensibility Support." Ada User Journal 35.2 (2014).

[20] Schmidt, Douglas C. "Guest editor's introduction: Model-driven engineering." Computer 39.2 (2006): 0025-31.

[21] Erlebach, Thomas; Jansen, Klaus (2001), "The complexity of path coloring and call scheduling", Theoretical Computer Science 255 (12): 3350, doi:10.1016/S0304-3975(99)00152-8, MR 1819065.

[22] Gandham, S.; Dawande, M.; Prakash, R. (2005), "Link scheduling in sensor networks: distributed edge coloring revisited", Proc. 24th INFOCOM 4, pp. 24922501, doi:10.1109/INFCOM.2005.1498534, ISBN 0-7803-8968-9.

ISBN 978-84-15914-12-9

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

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

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

[26] 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.

[27] George L., Appel A.W.: Iterated register coalescing. TOPLAS 18(3), 300324 (1996).

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

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

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

[31] Cooper, Keith D., and Anshuman Dasgupta. "Tailoring graph-coloring register allocation for runtime compilation." In Proceedings of the International Symposium on Code Generation and Optimization, pp. 39-49. IEEE Computer Society, 2006.

[32] Rosen, Barry K., Mark N. Wegman, and F. Kenneth Zadeck. "Global value numbers and redundant computations." In Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 12-27. ACM, 1988.

[33] Cytron, Ron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. "Efficiently computing static single assignment form and the control dependence graph." ACM Transactions on Programming Languages and Systems (TOPLAS) 13, no. 4 (1991): 451-490.

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

[35] Kosowski, Adrian, and Krzysztof Manuszewski. "Classical coloring of graphs." Graph colorings 352 (2004): 1-20.

[36] Welsh, Dominic JA, and Martin B. Powell. An upper bound for the chromatic number of a graph and its application to timetabling problems. The Computer Journal 10, no. 1 (1967): 85-86.

[37] Brlaz, Daniel. New methods to color the vertices of a graph. Communications of the ACM 22, no. 4 (1979): 251-256.

[38] Matula, David W., and Leland L. Beck. Smallest-last ordering and clustering and graph coloring algorithms. Journal of the ACM (JACM) 30, no. 3 (1983): 417-427.