Automatic Generation of Real-Time Aircraft Simulation System Configurations

EFRAIM JOSUE LOPEZ SANCHEZ

A Thesis

In The Department Of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements For the
Degree of Master of Applied Science (Electrical and Computer Engineering)
at

Concordia University
Montréal, Québec, Canada

May 2014

CONCORDIA UNIVERSITY
SCHOOL OF GRADUATE STUDIES

This is to certify that the thesis prepared

By: Efraim Josue Lopez Sanchez

Entitled: "Automatic Generation of Real-Time Aircraft Simulation System Configurations"

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science (Electrical and Computer Engineering)

Complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____ Chair
Dr. M. O. Ahmad

_____ Examiner, External
Dr. A. Awasthi (CIISE) To the Program

_____ Examiner
Dr. S. Abdi

_____ Supervisor
Dr. A. Hamou-Lhadj

Approved by: _____
Dr. W. E. Lynch, Chair
Department of Electrical and Computer Engineering

_____20_____

_____
Dr. C. W. Trueman
Interim Dean, Faculty of Engineering
and Computer Science

ABSTRACT

Building configurations for real-time aircraft simulation systems is a challenging task. It involves the distribution of the applications among different scheduling processes, bound to different CPU's, in such a way that the applications' priority and expected execution order are taken into account.

In this thesis, we report on a study conducted at CAE Inc., a world leading manufacturer of flight simulation products, in which we have developed an approach to automatically build configurations. Our approach is based on a greedy algorithm that uses heuristics to distribute many applications into different partitions in such a way that inter-partition communication is minimized, the load across partitions is balanced, and each partition is denoted as a binary tree (the data structure used by the scheduler to run the applications). The configuration is also constrained by the priority and execution time of the applications.

When applied to CAE, our approach produces configurations that in most cases outperform or are similar to those generated by a domain expert.

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my supervisor, Dr. Abdelwahab Hamou-Lhadj, for his guidance, comprehension and sincere support to define and develop my own research. He gave me meaningful insights and helped me see my research from different angles when I needed it the most. He complimented me on jobs well done and constantly motivated me to pursue excellence. More than anyone else, his influence has contributed to my research.

Much of this research was conducted at CAE Inc. I would like to thank Patrick Desrosiers, Martin Tapp, Patricia Gilbert, and many other CAE engineers for their help and knowledge. They helped me understand the most intricate and amazing details of real-time aircraft simulation systems. CAE also provided me with the tools and labs that I needed to evaluate the approach presented in this work. Without CAE support and involvement, this research could have not been possible.

I would also like to deeply thank CRIAQ (Consortium de recherche et d'innovation en aérospatiale du Québec), NSERC (Natural Sciences and Engineering Research Council of Canada), the Faculty of Engineering and Computer Science at Concordia University, CAE Inc., and Opal-RT for their financial support. This work would not have been possible without them.

Additionally, I would like to thank all members of the Software Behaviour Analysis (SBA) Research Lab at Concordia University for their friendship and encouragement. They brought the perfect atmosphere to combine work and life.

No matter where I go or what I become I will be always infinitely thankful to my parents. They gave me a life full of values and principles. Dad, from very early you taught me the importance of family, trust and love. Mom, you taught me to never give up and to follow my dreams. Every single thing I have achieved in my life is because of you both. I will never find enough words to tell you how much I love you. Thank you for all your sacrifices. Thank you for being such perfect parents. Love you!

Last but not least, I would like to express my gratitude to my awesome brother, Efraim Enmanuel. Thank you for spending tons of hours dealing with bureaucratic procedures in Venezuela so that I could be in a healthy financial position here in Canada. It made my life so much easier. I like to thank my oldest brother, Javier, for being a second father. I know I can always count on you no matter what. Finally, thanks to my sister Sara and my brother David. We all together make a wonderful family. Thank you all!

Table of Contents

List of Figures

List of Listings

List of Tables

Chapter 1 Introduction

1.1 Introduction to CAE's Simulation System

Flight simulators are essential components in aircraft industry. They are to reproduce, in a cost-effective way, the behaviour of all the elements of an airplane, its environment and the interactions between the airplane itself and its environment. To be cost-effective, most of the real aircraft's components are recreated by means of real-time simulation software applications. Flight simulation is used by the industry for a variety of reasons, the most important one is for pilot training. Other usages of aircraft simulators include the design and development of aircrafts.

A flight simulator is comprised of two major components: a mechanical part and a simulation unit. When a pilot interacts with the mechanical part, the latter produces electronic signals that are sent to the simulation unit through a middleware. The simulation unit does the required processing, and sends back the output to the mechanical part through the same middleware.

In this research, we are only concerned about the simulation unit, which in some ways, acts as the flight simulator's brain. It encompasses the real-time aircraft simulation applications that are in charge of replicating the behaviour of every single component of an aircraft. Some examples of these applications are the flight controls, autopilot, aircraft dynamics, cockpit, engine, radar, etc.

At CAE, the company on which this research was conducted, the simulation unit is comprised of many hosts, each of them running one or many schedulers. A scheduler is a process bound to a given CPU and is used to execute all possible applications registered in it. Each scheduler uses a binary tree data structure, also known as *the scheduler tree*, which provides an integration specialist the ability to define the execution order and priority of the applications involved in the simulation. Specifically, the scheduler's data is represented in an XML file. A CAE integration specialist uses this file to add applications to the corresponding scheduler tree. Applications could be registered in any scheduler tree node. During execution, each scheduler systematically traverses its scheduler tree, going through every tree path, giving each application access to the CPU to execute. The compendium of all the scheduler tree definitions done by the integration specialist is referred as the *flight simulation configuration files*.

1.2 Problem and Motivation

Simulation software is extremely complex by nature. This is usually comprised of hundreds or thousands of applications that work collaboratively to produce the final result. In the particular case of real-time aircraft simulation systems, such as the ones used at CAE, the situation is even more complex. These applications are low-latency applications that need to meet stringent timing constraints. They also have to execute with different priorities. This makes in most cases impossible to execute all the applications in a single processor.

To overcome this situation, it is necessary to partition the full simulation among a particular number of hosts or processors. Since applications have data dependencies, it is

important to distribute the simulation in different processors in such a way that inter-processor communication is minimized; while also keeping the total load balanced. Accomplishing the previous is necessary but not sufficient. For each resulting partition, it is still indispensable to arrange the applications in a binary tree so that their priorities and their expected execution orders are taken into account. As mentioned earlier, this binary tree serves as a configuration file that is used by the scheduler mechanism associated with each processor running the simulation.

Finding a global solution to the partitioning problem through exhaustive methods is unfeasible. The order of complexity of this approach would be $\boldsymbol{O(n^k)}$, where *n* is the number of applications and *k* the desired partitions. The combination of all possible solutions is giving by the Stirling Numbers of the Second Kind [Abramowitz72] (see equation (1)):

$$\boldsymbol{S_n^{(k)} = \frac{1}{k!}\sum_{i=0}^{k}(-1)^{k-i}\binom{k}{i}i^n\ ,\quad n \geq k \geq 1} \qquad \textbf{(1)}$$

This said, a heuristic-based approach is warranted. Aircraft simulation systems can be easily represented as a graph, where applications are nodes and their data dependencies are edges. The traditional way to tackle this issue would be through graph partitioning methods such as spectral clustering [Hagen92] and multi-level methods [Karypis95, Karypis98, Schloegel02]. However, to our knowledge, none of these techniques address the given constraints in our domain. They do not deal with must-link constraints, used to state that two or more nodes having a *special* relationship must be grouped into the same partition. This is essential to aircraft simulation systems as the core applications must run in a deterministic mode, and to ensure this, it is necessary, although not sufficient, that

these applications execute within the boundaries of the same processor, or *partition*. In addition, these techniques do not take into account any special semantic associated with the partitions. In our domain, each partition is abstracted as a binary tree, with finite capacity, and added semantic for defining applications' priority and execution order. To the extent of our knowledge, there is no algorithm in the literature that is designed to organize applications in a binary tree based on their priorities and expected execution order.

In the present time, at CAE, it is the responsibility of integration specialists to create scheduler configurations manually. They rely on knowledge acquired in the past to create ad-hoc configurations that are only valid for specific airplane models. In some cases, these configurations differentiate for the same model when they present small variations, for example, the same flight simulator that is adjusted twice to be sold to two different airlines. This process is error-prone and time consuming. To overcome this issue, we propose an approach that is intended to automatically develop configurations for real-time aircraft simulation systems used at CAE. First, the approach places all critical applications together in the same binary tree. Next, non-critical applications are placed in a systematic way. An application is placed in the current scheduler tree as long as it does not make the scheduler tree exceed its capacity; otherwise, the application is placed in the next one. Finally, a balancing step based on a generalized ratio-cut objective function [Yeh92] is used to minimize the dependencies among resulting scheduler trees, while maximizing the total load added to each of them.

1.3 Research Contribution

The main contributions of this thesis are as follows:

- An algorithm to automatically create configuration files for real-time aircraft simulation systems.
- The application of the approach to real world systems at CAE.
- The validation of the approach by comparing its effectiveness to configuration files created by domain experts.

1.4 Thesis Outline

The rest of the thesis is structured as follows:

- In Chapter 2, we present background information. In the first section, we contextualize CAE aircraft simulation architecture in further detail. We go through the concept of multi-host simulation architecture and explain how the scheduling mechanism works. Next, we introduce critical and non-critical overruns, two metrics used at CAE to evaluate the performance of the simulation. Then, we briefly comment on the qualification and approval process undergone by flight simulators, and its implications in the so called *Restricted-Loop*, a sequence of execution of the most critical applications. In the second section, we review the literature and we comment on the most influential works in the graph partitioning theory, and how they relate to our study.
- In Chapter 3, we propose an algorithm to automatically build configuration files for real-time aircraft simulation systems. We start the chapter with a brief

overview of it. Next, we explain in detail the multi-objective optimization problem that this algorithm aims at solving, and all the constraints it is subject to. The chapter proceeds with a formalization of all the inputs required by the algorithm. The proposed algorithm is a set of heuristics distributed in 3 major steps: a) placing critical applications; b) placing non-critical applications; and c) balancing; in the final section of this chapter we elaborate on these steps in further detail.

- In Chapter 4, we present an evaluation of our algorithm on CAE's flight simulation environment. We present the flight simulation unit technology used to test our algorithm. The chapter continues with a formal definition of the configuration (simulation units with 2 and 3 scheduling processes) and simulation scenarios (aircraft in air, on ground) used to carry out this experimental study. We ask an Integration Specialist to build the proposed configuration scenarios and we do the same using the algorithm. In the final section of this chapter we compare the results obtained by the human being versus the ones obtained with the algorithm.

- In Chapter 5, we summarize the main contributions of this thesis, and we comment on future directions.

Chapter 2 Background

2.1 Simulation Scheduling Mechanism

Real-time aircraft simulation systems are comprised of many applications that must run in a time-constrained fashion. To ensure that all of them run within the time constraint, it is necessary to spread the simulation among many processors. At CAE, this is performed following a trial-and-error process in which an integration specialist partitions the *N* number of applications into *K* number of partitions. Next, each group is assigned to a different scheduler, which will eventually execute the applications. Each scheduler runs as a separated process, and it is usually bound to a physical CPU in the target PC (see Figure 1). The drawback of partitioning the simulation is that a synchronization mechanism over the network must be activated. This adds a significant delay to the simulation if many applications running in one partition require data of many others running in a different partition.

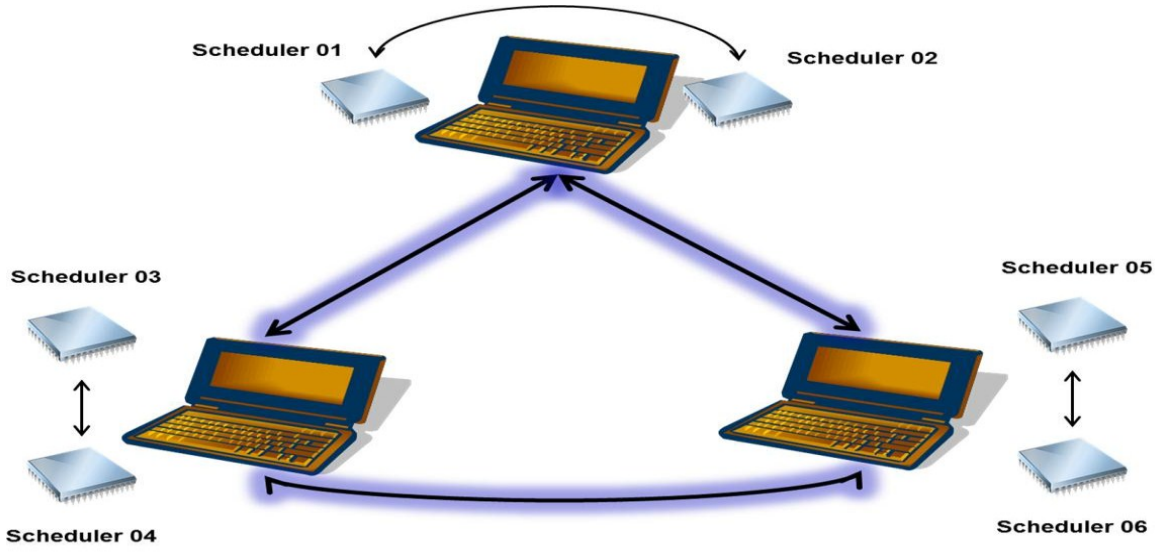

Figure 1. Multi-Host Simulation

Clustering the applications among many processors usually does not suffice. In each resulting partition, it is still necessary to provide a scheduling mechanism to ensure that applications not only run but also meet their expected execution rate (priority) and their expected execution order.

The scheduler uses a configuration file that enables an integration specialist to register the applications in a binary tree data structure (see Figure 2). Each tree node can hold zero or many applications, and each application is given an execution order within the node. Ultimately, the scheduler considers each tree path as an execution path. The root node is the first node of all execution paths, while any leaf node in the binary tree is the last one of its execution path. Note that there is a one to one relationship between a scheduler and its associated binary tree. For simplicity, in this work we will call the latter the scheduler tree.

The scheduler runs in an infinite loop, executing one tree path per cycle and within a time constraint. This is expressed in Hertz or milliseconds, usually 60Hz=16,7ms. A scheduler

tree of level 5, for example, means that the applications registered in the root node are executed approximately once every 16ms. Moreover, applications in second, third, fourth and fifth level are executed approximately once every 32, 64, 128 and 256ms respectively. It is easy to see that the root node always runs in each cycle, or at the fastest execution rate. As we will describe later, the root node contains the most critical applications.

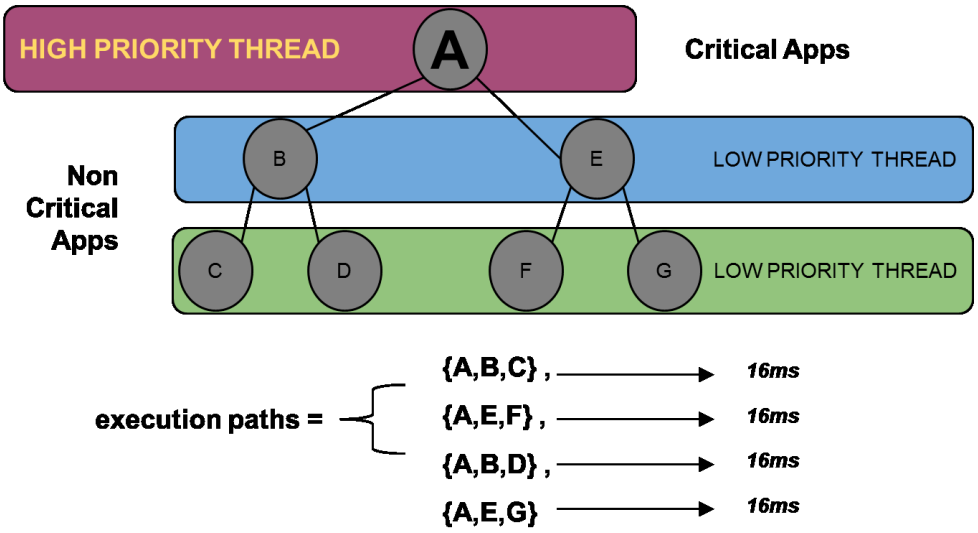

Figure 2. Scheduler Data Structure

2.1.1 Critical and Non-Critical Overruns

In an ordinary scenario, at each scheduler cycle, a high priority thread executes all applications registered in the critical node of an execution path, this is, the root node. Likewise, a low priority thread executes the applications registered in the remaining, non-critical, nodes. If the execution of the critical node does not complete within the time constraint, the scheduler waits for its finalization before calling the critical node of the next execution path. This is known as a *critical overrun*. If the execution of the non-

critical nodes does not complete within the time constraint, the scheduler pre-empts them and starts the execution of the next execution path. This is known as a *non-critical overrun*.

Critical and non-critical overruns can become problematic because they degrade the simulation and may result in an unusable flight simulator. An adequate configuration is the one that minimizes the number of overruns.

2.1.2 Qualification, Approval and Restricted-Loop

A flight simulator must be approved by the local national aviation authority. To do so, a number of tests are executed against the simulator and the results are evaluated based on a predefined level of criteria. In the particular case of the United States of America, these qualification criteria and regulations are imposed by the Federal Aviation Administration (FAA) through 14 CFR Part 60 [FAA06]. One of the key points to evaluate in this qualification and approval process is the delay time between critical components of an airplane. In the attachment 2 to Appendix A to Part 60, FFS Objective Tests, Paragraph #15, we find the title Transport Delay Testing, which literally states the following:

"The transport delay should be measured from control inputs through the interface, through each of the host computer modules and back through the interface to motion, flight instrument, and visual systems. The transport delay should not exceed the maximum allowable interval"

To comply with the aforementioned, an integration specialist must ensure two aspects. First, all critical applications must run according to a governing execution order. This is

also called the *Restricted-Loop*. Second, all critical applications must run within the same processor, hence registered in the same scheduler tree. These two aspects convey to the required determinism.

The restricted-loop can be defined as a set of ordered categories. Note that each critical application belongs to exactly one category defined in the loop. The arrangement of the categories in the loop inherently imposes the final execution order of the critical applications. Applications belonging to the same category can execute in any order within the boundaries of their categories. However, they can execute only after applications in the preceding category complete. Examples of categories defined in the loop are: flight controls, aircraft dynamics, flight instruments and visual and motion cueing. For example, to comply with the regulations, all flight control applications must execute before the aircraft dynamics ones, and visual and motion cueing applications can execute in parallel right after the aircraft dynamics ones run. It is important to highlight that non-critical applications are not constrained by this, so they can be placed in any cluster and scheduler tree node that leads to good results.

2.2 Related Work

Scientific simulation systems can be easily represented as a graph, where applications are nodes and their relationships are edges. There will be a relationship between two applications A and B if A produces an output that serves as input for B. The problem of partitioning a graph has been studied for years. This is an NP-Hard problem, hence finding a global optimal solution to it is intractable. Many heuristics coming from all

sorts of disciplines in computer science have been suggested. In the next subsections, we present the state of the art of the most common graph partitioning methods.

2.2.1 Mincut Methods

Many minimum cut algorithms have been proposed. The main idea is to obtain two partitions A and B out of a graph G such that the edge-cut, the weight of the edges linking the nodes between A and B, is minimized [Jain10]. One of the first efforts was proposed by Kernighan et al [Kernighan70], which produces local optima. The idea is to start with an initial partition and then continuously swap the two applications that reduce the most the edge-cut between partitions. The problem with this technique is that it strongly depends on the initial partition. Karger [Karger93] proposed to randomly collapse the nodes of a graph until only two nodes are left. In general, pure minimum edge-cut algorithms may easily lead to imbalanced partitions.

2.2.2 Spectral Clustering Methods

Spectral graph theory has been used to address the partitioning problem. Spectral clustering approaches rely on properties of the entire graph and may yield global optima. A matrix of pairwise similarities between the nodes is built, from which an adjacency and a degree matrix are derived, which in turn are used to create the laplacian matrix. Next, the first K eigenvalues and eigenvectors of the laplacian are computed, where K equals the number of desired partitions. The final step is to arbitrary choose a clustering algorithm, e.g. *K-means*, and produce the K partitions based on the obtained eigenvectors [Luxburg07]. An imbalanced partition is one of the major drawbacks of the initial methods following this approach as their objective functions are based on the minimum

cut. Hagen et al. [Hagen92] proposed probably one of the most influential works. They used the Ratio Cut metric introduced by Wei et al. [Wei89] as objective function, which favors for more balanced partitions.

Shi et al. [Shi00] used spectral clustering to address the image segmentation problem. The problem is represented as a weighted undirected graph *G=(V,E),* where the weight of the edges *w(i,j)* is based on a function that measures the similarity between the nodes *i* and *j*. The goal is to partition the graph V into disjoints V_1, V_2, …, V_i, so that the similarity between nodes belonging to different groups V_i, V_j is low, and high among nodes within the same set V_i. To solve this issue, a normalized cut cost function is proposed. The function calculates the cost as a fraction of the similarity between the nodes in the graph. This way, partitions with small number of nodes will certainly have a big cut value, hence discarded. Next, the cost function is minimized by computing a generalized eigenvalue problem.

2.2.3 Multi-Level Methods

Most recently, multi-level methods have gained some relevance. Without loss of generality, the partitioning is done in 3 phases [Karypis95]. First, the coarsening phase permits to reduce the complexity of the graph by collapsing its nodes in an iterative process. Each iteration results in a graph that is smaller than the previous one. Different matching criteria are used to merge the nodes, being Random Matching (RM), Heavy Edge Matching (HEM), Light Edge Matching (LEM) and Heavy Clique Matching (HCM) the most commons. Secondly, the partitioning phase takes place. The central idea is to obtain a minimum edge-cut of the coarsest graph such that the total nodes' weight is

balanced among the resulting partitions. The partition can be done using different approaches: Spectral Bisection, Combinatorial Methods, or even Geometric Bisection. Finally, in the uncoarsening phase the coarsest graph is mapped back to the original one. Karypis et al. [Karypis95] proposed METIS, probably the most influential work following this philosophy.

2.2.4 Evolutionary Methods

Genetic algorithms, a.k.a GA's, are heuristics based on the theory of natural selection or evolution that are used in a wide range of fields in computer science to mainly address optimization and search problems. The population, which are candidate solutions to the problem, is subject to crossover and mutator operators that make it change over time. In the next generation of the population, the less fit individuals, among parents and offspring, are left out based on a fitness function. More specifically, genetic algorithms have been used to tackle the graph partitioning problem. Maini et al. [Maini94] used crossover operators that take advantage of domain specific knowledge and information from history of genetic search. Each individual, or candidate solution, is represented as a vector in which the i^{th} element is mapped to a specific node in the graph and its value in the vector matches its target partition. The fitness function considers both the inter-partition communication cost and the partitions load balance. The crossover operator is based on a probability vector that is used to select the best genetic material from both parents, or from the region that is best known to produce higher quality individuals. One of the drawbacks of genetic algorithms is that they might get stuck in local optimal solutions. Besides, they require much more time to execute when compared with multi-

level, spectral or mincut methods, and the solution depends on an initial set of individuals that are randomly generated in most cases.

2.2.5 Discussion

To the extent of our knowledge, there is no graph partitioning method that permits adding must-link constraints, though this has been already studied in *Constrained Clustering* through different implementations of *K-means*. This is particularly relevant to our problem since we need to place all critical applications, or the Restricted-Loop, within the same partition. Additionally, as far as we know, there is no algorithm designed to arrange components within a binary tree considering constraints such as priorities and sequence of the components, as well as the total load added to the tree paths.

Chapter 3 Algorithm

3.1 Overview

In this section, we introduce our approach to automatically develop configurations for real-time aircraft simulation systems. The approach encompasses three steps (see Figure 3).

In the first step, we arrange the critical applications in the same scheduler tree, while respecting the order of applications based on their categories and the restricted-loop. In the second step, we place the non-critical applications by filling up the capacity of each scheduler tree. It is only after one scheduler tree is complete that we move to another scheduler tree. In the final step, we balance the scheduler trees using a generalized ratio cut objective function [Yeh92]. The idea is to minimize the dependencies among scheduler trees while keeping the load of each partition balanced.

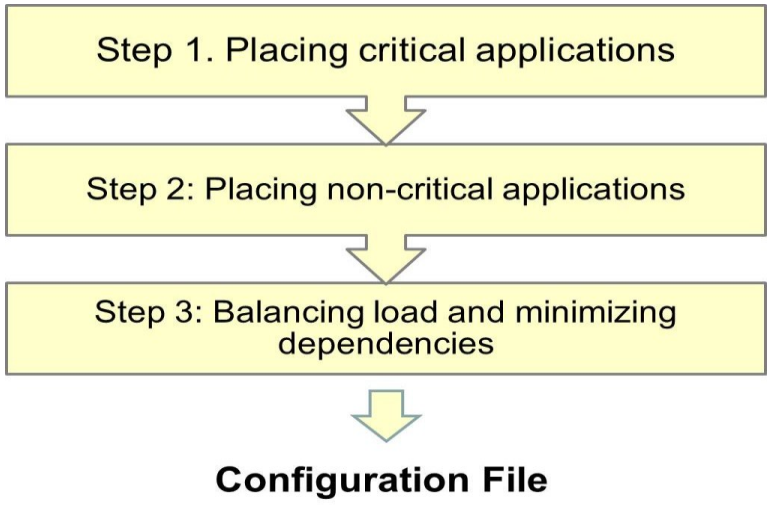

Figure 3. Algorithm's general approach

3.2 Objectives and Constraints

This work intends to solve an optimization problem whose general objectives are:

a) Distribute the applications among hosts while minimizing inter-host dependencies.

b) Distribute the applications among hosts while keeping the load balanced.

c) In each resulting scheduler tree, distribute the applications among execution paths while keeping the load balanced.

While the first two optimization objectives are tackled in the last step of the algorithm, the last one is met every time an application is placed in a scheduler tree through common functionalities.

Note that our work is also constrained by the definition of the restricted-loop exposed in the previous section:

d) Critical applications must run within the same partition.

e) Critical applications must run by respecting the governing order.

Two additional constraints that apply generally to all applications are also added:

f) Applications should run at their expected execution rate (priority).

g) Total path execution time cannot exceed an establish threshold.

The aforementioned time threshold defines the time constraint for each execution path. If a scheduler runs at a frequency of 60Hz, this means that a tick is generated approximately

every 16.7ms. Upon receipt of a tick, the scheduler should run the next execution path. In other words, the current running execution path has 16.7ms to complete.

3.3 Algorithm Inputs

Several inputs must be provided to the algorithm so that its goals are met. This section formally defines each of them.

3.3.1 Applications Set

These are the applications that run the simulation. Each of them has to be registered in one scheduler tree. Let's APPS be the set of non-ordered applications, this is:

$$APPS = \{app_1, app_2, app_3, \ldots, app_i\} \quad (2)$$

Where *i* is said to be the total number of applications to be considered by this algorithm and app_i is a given application in the set. APPS cannot be empty. Each *app* in APPS is a 3-tuple denoted as:

$$app = (ExecRate, ExecTime, Category) \quad (3)$$

ExecRate and *ExecTime* are the expected execution rate (priority) and the expected execution time in Hertz. The last one defines the category to which the application is associated.

3.3.2 Categories

To differentiate between critical and non-critical applications, we check if the application's category is in the restricted-loop. In such a case, we say that the application is critical; otherwise it is non-critical. For instance, an application "A" is linked to a

category “C”. If this category is in the restricted-loop, then “A” is critical. Let CATS be the set of non-ordered categories, this is:

$$CATS = \{cat_1, cat_2, cat_3, \ldots, cat_i\} \quad (4)$$

Where *i* is said to be the total number of categories to be considered by this algorithm and cat_i is a given category in the set. CATS cannot be empty as it must contain at least a *default* category, which is normally assigned to a non-critical application. Each *cat* in CATS is denoted as:

$$cat = (Name) \quad (5)$$

Name refers to the unique, and self-descriptive, identifier of the category.

3.3.3 Dependency Graph

Applications share information by exchanging data. For example, an application “B” depends on “A” if “A” modifies or overwrites the content of a variable “V”, used later by “B”. For instance, the engine system and the fuel indicator are dependent on each other. As the engine consumes oil, after each execution it has to overwrite the new available oil quantity. Then, this new quantity is provided as input to the fuel indicator so that the pilot can see updated information. If these two applications “A” and “B” happen to run in different hosts, a synchronization process runs so that the new variable values are transferred at each scheduler cycle.

We represent these dependencies using a dependency graph, and the algorithm uses it to minimize this problem. Let DEPENDENCIES be a directed graph, which can be represented as an ordered pair:

$$DEPENDENCIES = (S, T) \quad (6)$$

Where:

- S is a set of vertices, each one representing an application.
- T is a set of ordered pairs of vertices, called edges, weighted, and indicating a dependency between two applications.

In a directed edge $E = (x, y, w)$, y is said to be dependent on x with weight w.

3.3.4 Restricted-Loop

As mentioned before, this is a list of ordered categories that defines the execution order of critical applications. In the loop, a given category may be succeeded or preceded by one or many categories. For instance, the following draws what could be a valid sequence: A→B; A→C; B→D; C→D. In this example, "D" can only execute after "B" and "C". Likewise, "B" and "C" can only execute after "A" completes. To formalize this, let RESTRICTED_LOOP be a directed acyclic graph (DAG), which can be represented as an ordered pair:

$$RESTRICTED\_LOOP = (V, A) \quad (7)$$

Where:

- V is a set of vertices, each one representing a category.
- A is a set of ordered pairs of vertices, called edges, non-weighted, indicating a sequence of execution between two categories.

In a directed edge $E = (x, y)$, y is said to be the successor of x, and x is said to be the predecessor of y. To traverse this graph, the starting vertex, or root element, must also be provided.

3.3.5 Number of Desired Partitions / Scheduler Trees

As mentioned in Sections 2.1, there is one scheduler running per CPU, and each scheduler has an associated scheduler tree data structure. In this sense, the number of desired partitions equals the number of desired schedulers, hence, the number of required scheduler trees. Although the user only provides the number of desired partitions, it is worth mentioning that our algorithm internally represents this as *initially* empty scheduler trees. Let SCHEDULERS be a set of non-ordered trees, this is:

$$\text{SCHEDULERS} = \{\text{tree}_1, \text{tree}_2, \text{tree}_3, \ldots, \text{tree}_i\} \quad (8)$$

Where i is said to be the total number of schedulers or partitions desired and $tree_i$ is a given tree in the set. SCHEDULERS cannot be empty. Each *tree* in SCHEDULERS is a directed acyclic graph, which can be represented as an ordered pair:

$$\text{TREE} = (V, A) \quad (9)$$

Where:

- V is a set of vertices, each one representing an ordered list of applications.
- A is a set of ordered pairs of vertices, called edges, non-weighted, indicating a relationship between two nodes.

Note that this is a binary tree. In others words, at each level each vertex has exactly two directed edges, or children, except for the last level where vertices have no children. Each vertex *v* in V is denoted as:

$$\text{VERTEX\_APPS} = (app_1, app_2, app_3, \ldots, app_i) \quad \textbf{(10)}$$

Where *i* is said to be the total number of applications registered in the node and app_i is a given application in the node. VERTEX_APPS can be empty. Each *app* in VERTEX_APPS is a subset of the previously defined APPS, this is:

$$\text{VERTEX\_APPS} \subseteq \text{APPS} \quad \textbf{(11)}$$

Initially, every VERTEX_APPS is empty. As the algorithm advances, applications will be registered.

3.3.6 Number of Scheduler Tree Levels

An application execution rate is guided by the level of the node which it is registered in. Nodes belonging to each next level in the tree execute at a double rate than nodes in the previous level. For example, assuming that the scheduler is running at a frequency of 60 Hertz, this means that the root node, level 1, executes at 60 Hertz. Next nodes, those in the second level, execute at 30 Hertz, followed by those in level 3 at 15 Hertz, and so on. Applications registered in the root node execute at the fastest rate. Likewise, applications registered in any leaf node execute at the slowest rate. Note that a given application is registered at the fastest closest available execution rate. If the expected execution rate of an application “A” is 25 Hertz, in the previous example it will be registered in level 2.

The required level of the scheduler trees is provided as input as SCHEDULER_TREE_LEVELS.

3.3.7 Execution Path Real Time Budged (Scheduler Frequency in Hertz)

As pointed out in Section 2.1, each tree path is considered as an execution path for the scheduling mechanism. The frequency at which the scheduler is running imposes the real time constraint. For instance, if a scheduler is running at 60 Hertz, this means that a tick is generated approximately every 16.7ms. Upon receipt of a tick, the scheduler should execute the next execution path. In other words, an execution path has 16.7ms to complete. This input is expressed in Hertz and is denoted as EXEC_PATH_REAL_BUDGET.

3.3.8 Execution Path Virtual Time Budget

In practice, placing applications in a given execution path until it is at capacity (real time budged) may not be the best way to proceed. At any time, the operating system may switch the scheduler (do not confuse with the operating system scheduler) from running state to ready or waiting state in response to an interrupt or simply to give another process an opportunity to execute. Due to this, we cannot take the real time budget for granted. We use instead a percentage of the time budget to constraint the load added to an execution path by the algorithm. This input is expressed as a real number $\in [0, 1]$ and is denoted as VIRTUAL_TIME_PERC. For example, if the real time budget is 60 Hertz and this input is 30% (expressed as 0.30), this means that the algorithm will add approximately up to 5ms load to each execution path:

$$\frac{1000ms}{60Hz} * 0.30 = 5\text{ms} \quad (12)$$

3.4 Algorithm Steps

To meet the objective in Section 3.2.C and the general constraints in Sections 3.2.F and 3.2.G, we have developed algorithm functionalities that are used by all the steps. More specifically, to register an application in a scheduler tree we first ascertain the target execution level of the application. Considering that the root node is executing at the given (input) scheduler frequency, and that each next level runs at half rate of its preceding level (see Sections 2.1 and 3.3.6), the application execution level is given by the following:

$$level = \left\lfloor log_2\left(\frac{Scheduler\ Frequency\ in\ Hertz}{App\ execution\ rate\ in\ Hertz}\right) + 1 \right\rfloor \quad (13)$$

The floor function is necessary because we need to approximate the application execution level to the closest fastest available execution level. For instance, let us assume that we are dealing with a scheduler tree of 3 levels, running at 60Hz, 30Hz, and 15Hz respectively. In this scenario, the target execution level of an application whose execution rate is 25Hz would be the second level, running at 30Hz, and not the third one, running at 15Hz. It is important to point out that an application execution rate cannot be higher than the scheduler frequency. Also note that we add one just to make level base 1. We do this only for the sake of clarity. Level base 0 may be confusing.

The next step is to find the candidate vertex in the obtained level that can support the load without affecting the balance of the whole tree. The best candidate vertex is the one that:

- All possible execution paths crossing the vertex have enough room to accommodate the additional weight associated with the application. No execution path exceeds the time constraint. We call this a *feasible vertex*.

- Among all feasible vertices, the total load summation of all paths crossing the vertex is the minimum.

More formally, the total load added to a vertex is given by:

$$\boldsymbol{vertex.ExecTime} = \sum_{\boldsymbol{i=1}}^{|\boldsymbol{APPS\ IN\ VERTEX}|}(\boldsymbol{app_i.ExecTime}) \quad \textbf{(14)}$$

Likewise, the total load added to an execution path is given by:

$$\boldsymbol{path.ExecTime} = \sum_{\boldsymbol{i=1}}^{|\boldsymbol{VERTICES\ IN\ PATH}|}(\boldsymbol{vertex_i.ExecTime}) \quad \textbf{(15)}$$

Then, for each vertex in the previously found tree level, the total load summation of all paths crossing the vertex is given by:

$$\boldsymbol{vertex.G\_ExecTime} = \sum_{\boldsymbol{i=1}}^{|\boldsymbol{PATHS\ CROSSING\ VERTEX}|}(\boldsymbol{path_i.ExecTime}) \quad \textbf{(16)}$$

Where, $\forall\, p \in paths\ crossing\ vertex \mid p_i.ExecTime + app.ExecTime \leq (EXEC\_PATH\_REAL\_BUDGET * VIRTUAL\_TIME\_PERC)$

If vertex meets the previous constraint, then vertex is in the *feasible* set. Lastly, the best candidate vertex is given by:

$$\boldsymbol{best\_vertex} = \mathbf{min}(\forall\, \boldsymbol{f} \in \boldsymbol{feasible} \mid \boldsymbol{f_i.G\_ExecTime} < \boldsymbol{f_j.G\_ExecTime}) \quad \textbf{(17)}$$

Finally, if *best_vertex* exists, then the given application is registered there at the last position and TRUE is returned. Otherwise, FALSE is returned.

The next two figures are examples of the procedure used by the algorithm to register an application. In figure 4 a scheduler tree of 3 levels is depicted. The head, or root, node is running at 60Hz. Assuming a virtual capacity of 100% of the theoretically real capacity (see Section 3.3.8), the time constraint of every single execution path is approximately 16.67. This is: 1000s / 60Hz. We want to register an application that needs 12ms to run and is expected to be executed at a rate of 30Hz. If the root node is running at 60Hz, this means that the second level is running at 30Hz. That said, the target execution level for this application would be 2. As depicted in the figure 4, there are 2 available nodes at the target level. We can see that before trying to register the application there is already an accumulated load of 5ms in the level 2's second node. Adding the expected application load, this is 12ms, to this node would be unfeasible as this would cause the 2 execution paths crossing the node to have a total load of 17ms, which clearly exceeds the time budged. As it can be noticed, the only feasible node is the level 2's first node; consequently, the application is registered in it. The scenario after registering the application is depicted in the right part of Figure 4.

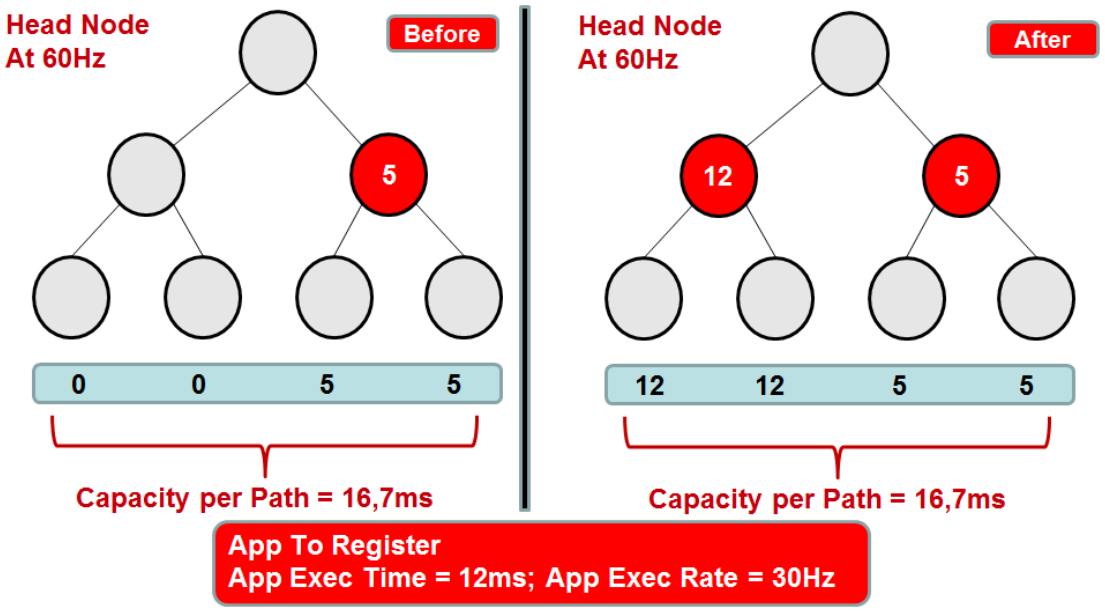

Figure 4. Registering application procedure (1/2)

Next, using the same scheduler tree, we want to register an application that needs 3ms to run and is expected to be executed at a rate of 15Hz. In Figure 5, it is not difficult to see that the target application execution level is 3, and there are 4 available nodes at this level. Adding the expected application load to any node in the third level will not cause any problem. This means that the four execution paths are feasible. In this case the best candidate node is the one whose summation of all paths crossing it is the minimum. Registering the application in either first or second node of Level 3 would result in execution paths with 15ms. On the other hand, registering the application in either level 3's third or fourth node would result on execution paths with 8ms. It is clear that the best candidate node is either the third or the fourth. To be systematic, we simply pick the first one from left to right among the best options. The scenario after registering the application is depicted in the right part of Figure 5.

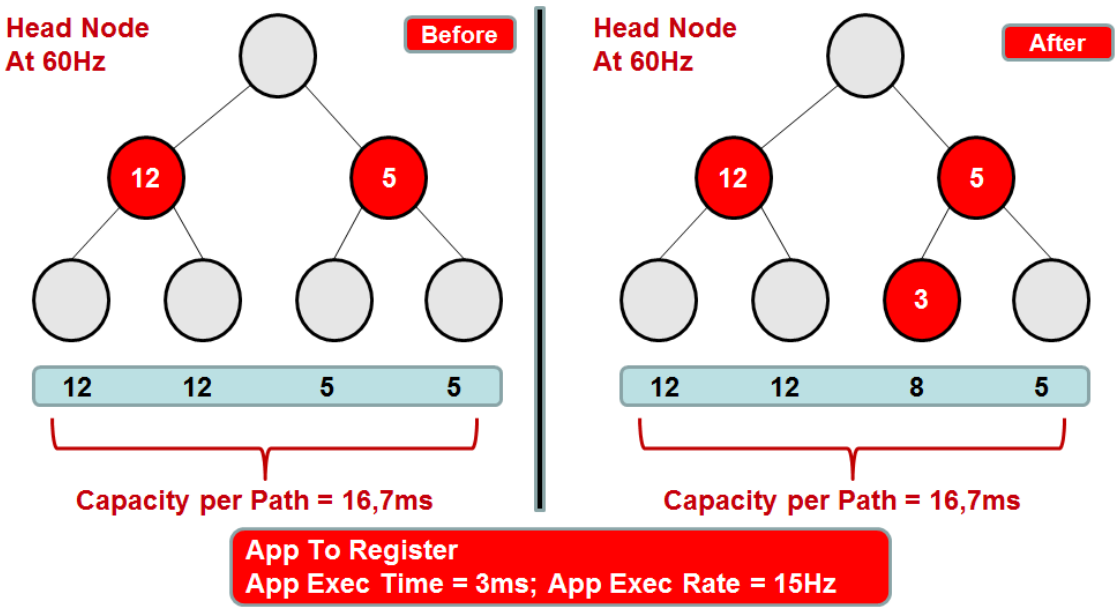

Figure 5. Registering application procedure (2/2)

Finally, the pseudo code for the *registerApp(schedulerTree,app)* function and its related functions is shown below in listings 1, 2 and 3.

```
function registerApp(schedulerTree, app):
1:  level←targetDepthLevel(app.ExecRate)
2:  vertex←bestFeasibleVertex(schedulerTree, level, app.ExecTime)
3:  if vertex not NULL then:
4:      addAtLast(vertex, app)
5:      return TRUE
6:  else:
7:      return FALSE
8:  end if
```

Listing 1. Pseudo-Code for registering an application in a given tree

```
function targetDepthLevel(execRate):
 1: if execRate > SCHEDULER_FREQUENCY then:
 2:     /* ERROR, APP CANNOT RUN FASTER THAN SCHEDULER*/
 3: end if
 4: log2←log base 2 (SCHEDULER_FREQUENCY / execRate) + 1
 5: targetLevel←floor(log2)
 6: if targetLevel <= SCHEDULER_TREE_LEVELS then:
 7:     return targetLevel
 8: else:
 9:     return SCHEDULER_TREE_LEVELS
10: end if
```

Listing 2. Pseudo-Code for finding an application's target tree level

```
function bestFeasibleVertex(schedulerTree, level, execTime):
 1: bestVertex←NULL
 2: for each vertex in VERTICES IN (level) do:
 3:     feasible←TRUE
 4:     vertex.G_ExecTime←INFINITY
 5:     for each path in PATHS CROSSING VERTEX do:
 6:         if path.ExecTime + execTime > TIME BUDGET then:
 7:             feasible←FALSE
 8:             /* NON FEASIBLE VERTEX, BREAK INNER LOOP */
 9:         else:
10:             vertex.G_ExecTime←vertex.G_ExecTime+path.ExecTime
11:         end if
12:     end for
13:     if feasible and vertex.G_ExecTime<bestVertex.G_ExecTime then:
14:         bestVertex←vertex
15:     end if
16: end for
```

```
17: return bestVertex
```

Listing 3. Pseudo-Code for finding the best feasible node for an app in a given tree and level

3.4.1 Placing Critical Applications

The idea behind placing critical applications is simple. We need to ensure that there exists a scheduler tree with enough room to accommodate all critical applications together (see Section 3.2.D). As a convention, we call this tree the *critical scheduler tree*. Note that the critical scheduler tree could be any of the desired scheduler trees (see Section 3.3.5). If we were to place critical and non-critical applications in the same step, it is easy to see that the load carried by non-critical applications may fill up the critical scheduler tree before having placed all critical applications. Some may argue that it is still possible to open room for critical applications by moving non-critical ones to a non-critical scheduler tree. This has particularly two disadvantages:

- It may easily lead to a cumbersome step.
- Ideally, we want to move applications that degrade the performance the least (see Section 3.2.A). Moving applications among scheduler trees before having the full picture may lead to wrong local optimal decisions.

To meet the constraint stated in Section 3.2.E, we use the well-known *Breadth First Search* algorithm [Cormen09], or just BFS, to traverse the categories of the RESTRICTED_LOOP (remember that the loop is a DAG and each vertex represents a category; see sections 2.1.2 and 3.3.4). Using BFS ensures that the algorithm traverses all the vertices that are at a distance *N* from the starting vertex of the RESTRICTED_LOOP before traversing those at a distance *N+1*. Upon traversing, or visiting a category "C", the

algorithm queries all the applications in APPS (see Section 3.3.1) whose associated category matches "C". Next, all found applications are registered in the critical scheduler tree. If trying to place a critical application in the critical scheduler tree makes the latter overflows its capacity, then there is no feasible solution to this problem. In such case, the algorithm aborts. Otherwise, this step stops when all categories have been processed. The pseudo code for this step is shown in Listing 4.

```
function placeCriticalApps(configuration):
1:  schedulerTree← find critical scheduler tree
2:  categories←BreadthFirstSearchIterator(RESTRICTED_LOOP)
3:  for each category in categories do:
4:      appsInCategory←APPS.findAppsByCategory(category)
5:      for each app in appsInCategory do:
6:          ok←registerApp(schedulerTree, app)
7:          if ok==FALSE then:
8:              abort algorithm
9:          end if
10:     end for
11: end for
```

Listing 4. Pseudo-Code for the Placing Critical Applications Step

3.4.2 Placing Non-Critical Applications

At this point all critical applications have been already registered in the critical scheduler tree. In this step, we apply heuristics aiming at placing highly dependent applications together in the same scheduler tree. We first need to define an initial pivot application that is already registered in the current scheduler tree, which initially is the critical scheduler. The pivot application is subsequently used to find highly related applications

and register them in the current scheduler tree. We could choose any arbitrary application among the already registered critical applications. However, this would only allow us to place applications that are relevant to the chosen critical application. Instead, we build a *collapsed* version of the DEPENDENCIES graph (see Section 3.3.3). The collapsing strategy is straightforward. We merge all critical application vertices in DEPENDENCIES. We call this new vertex the *critical set vertex*. At this point we are able to choose this critical set vertex as pivot application. Following this strategy allows us to find relevant applications to the whole critical set of applications. This is interesting because in reality the critical set of applications can be seen as a whole unit that cannot be separated. And even more importantly, we prioritize non-critical applications that highly influence the behaviour of the whole RESTRICTED_LOOP. An example of the collapsing strategy is shown in Figure 6. The original dependency graph is depicted on the left. All critical applications are coloured in red and non-critical ones in white. The new collapsed version of the dependency graph is depicted on the right. The new resulting node in red is the *critical set vertex*.

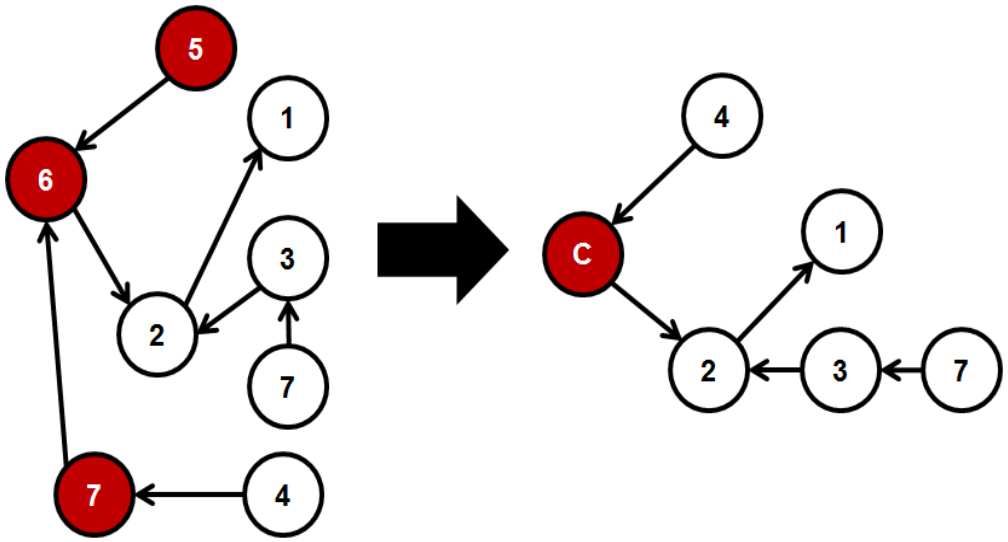

Figure 6. Collapsing strategy example

Next, we enter in a loop and at each iteration we register the applications in the current scheduler tree that are not yet marked as registered and are related to the pivot application. The order in which these applications are registered is given by their relevance to the pivot application. To measure this relevance we simply query all the adjacent vertices of the pivot application in the collapsed dependency graph. Then, these edges are sorted in descending order according to their weights. Note that once an application is registered, it is marked as so and consequently push onto a queue. When all applications related to the pivot applications have been processed, a new pivot application is taken from the queue and this process starts all over again. This is particularly important because the next pivot application is the next one in order of relevance to an already registered application. The assumption is ordinary: applications that are highly relevant to the new pivot application are likely to be highly relevant to one or many applications already registered in the current tree. If the queue is empty and there are applications yet to be placed, this means that we were in the presence of a disjoint group of applications. This situation is exceptional and unlikely to happen. In such cases,

we choose a new pivot application among the not marked ones. The new pivot would be the one whose sum of inbound and outbound edge weights is the maximum. The reason is that this application is likely to be the most representative one, and consequently more likely to pair relevant applications together within the same scheduler tree.

At some point trying to place an application in the current scheduler tree will return FALSE since it does not have enough room to accommodate it. In such cases we assign the next available tree in SCHEDULERS (see Section 3.3.5) to the *current* pointer and try to register the application again. For example, if *current* tree is in the 2nd position out of 3 trees in SCHEDULERS, the next available tree is the one in the 3rd position. Likewise, if *current* tree is in the 3rd position, the next available tree is the one in 1st position. We keep trying moving to the next available tree until *registerApp(schedulerTree,app)* (see section 3.4) returns TRUE or all of the trees in SCHEDULERS return FALSE. If we cannot register the application in any tree, then the algorithm cannot find a solution to this problem. In such case it aborts. As the reader can realize this step does not tackle any objective or constraint. Nonetheless, it prepares a good initial partition for the next step. The pseudo code for this step is shown in listing 5.

function placeNonCriticalApps():

```
1:  Q←create a queue
2:  currentTree ← find critical scheduler tree
3:  G←collapse critical apps in DEPENDENCIES
4:  pivotApp ← find critical set vertex in G
5:  while pivotApp is not NULL do:
6:      mark pivotApp as visited
7:      edges←G.adjacentEdges(pivotApp)
```

```
8:      sort edges from highest to lowest weight
9:      for each e in edges do:
10:         app←e.oppositeVertex(pivotApp)
11:         if app is visited then:
12:             continue with next edge
13:         end if
14:         ok←register app in current or next available scheduler tree
15:         if ok==FALSE then do:
16:             abort algorithm
17:         end if
18:         mark app as visited
19:         enqueue w onto Q
20:     end for
21:     pivotApp←Q.dequeue()
22:     if pivotApp is NULL then:
23:         pivotApp←find not marked vertex with highest inbound outbound
        edge  weight sum in G
24:         ok←register pivotApp in current or next available scheduler tree
25:         if ok==FALSE then do:
26:             abort algorithm
27:         end if
28:     end if
29: end while
```

Listing 5. Pseudo-Code for the Placing Non-Critical Applications Step

3.4.3 Balancing

In this final step, we intend to minimize the number of dependencies among applications belonging to different scheduler trees, or the inter-cluster dependencies, while also balancing the total load of the resulting clusters (see Sections 3.2.A and 3.2.B). To tackle this optimization problem, in this work we use a generalized k-way ratio-cut cost

function, also called the cluster ratio, introduced by Yeh, Cheng and Lin [Yeh92]. Our objective is to minimize this cost function:

$$\boldsymbol{cost} = \frac{(\sum_{i=1}^{k} \mathbf{E_i}) + \mathbf{1}}{\sum_{i=1}^{k-1} \sum_{j=i+1}^{k} |\boldsymbol{tree_i}| * |\boldsymbol{tree_j}|} \quad \textbf{(18)}$$

Where:

- k equals the number of desired partitions, this is: |SCHEDULERS|
- $tree_i$, or $tree_j$, is a given tree in SCHEDULERS
- E_i equals the sum of the weights of directed edges with source in $tree_i$ and target in $tree_j$, for $i \neq j$. Note that we could sum all inbound and outbound edges. In such case the edge cut (numerator), which is the sum of the weights of the edges in the cut, would be: $(\frac{1}{2} \sum_{i=1}^{k} E_i) + 1$
- We add 1 to ensure a minimum cut of 1. A value of 0 cancels the numerator.

The total load of a given scheduler tree is given by the sum of the load added by each of the applications registered in the tree. Likewise, the total load that an application adds to its tree is given by its total execution time times the number of paths that intersect the scheduler tree node which it belongs to. This is:

$$\boldsymbol{load} = \sum_{i=1}^{|\boldsymbol{APPS\ IN\ SCHEDULER\ TREE}|} \left(\boldsymbol{app_i.ExecTime} * \mathbf{2}^{\boldsymbol{MAX - app_i.ExecLevel}}\right) + \mathbf{1} \quad \textbf{(19)}$$

Where:

- MAX equals the input SCHEDULER_TREE_LEVELS.

- $app_i.ExecLevel$ is the application execution level as defined in the equation (13) in the section 3.4.

- We add 1 to ensure a minimum weight of 1. A value of 0 cancels the denominator in the cost function (18), producing an invalid division by 0.

We do not use the application execution time alone to compute the scheduler tree load due to an important reason. A given application "A" could be registered in one or many execution paths, and each execution path is independently executed. If this same application "A" were to be registered, say, in 4 execution paths, "A" would be executed in 4 different scheduler iterations. We would be ignoring this fact if we were to count the execution time of "A" only once. To illustrate, assume there are two scheduler trees T1 and T2 of 3 levels each. An application A1 is registered in the root node of T1. And applications B1, B2, B3, and B4 are registered in each leaf node of T2. All applications execute in 2ms. In this example all execution paths in T1 and T2 execute in 2ms, even though only 2ms were added to T1 and 8ms were added to T2. In terms of time required to execute, T1 and T2 are equivalent. This exemplary scenario is depicted in Figure 7.

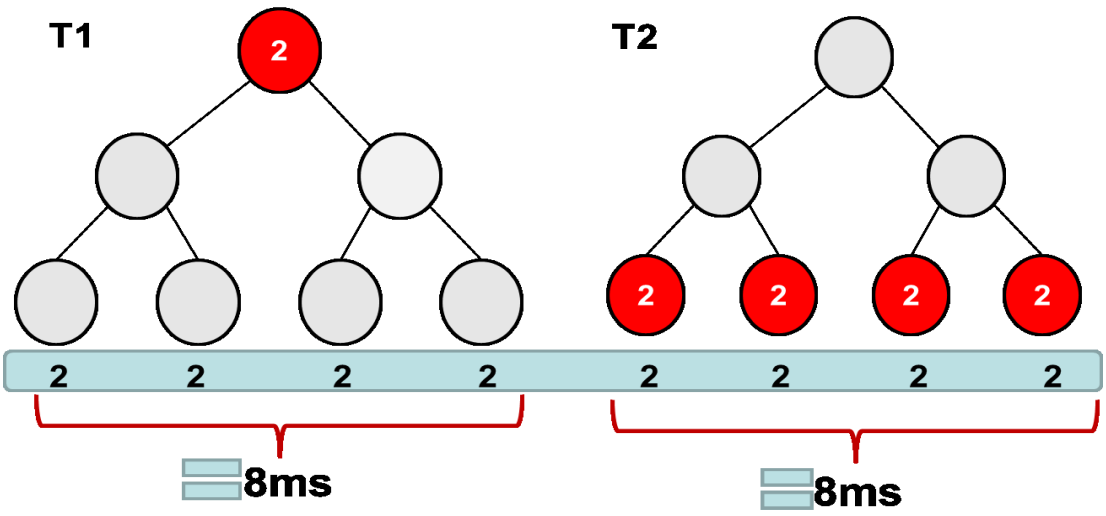

Figure 7. Scheduler tree load example

To demonstrate how the cost function leads to good results, let us analyze the example depicted in Figure 8. For simplicity, all the vertices, each of them corresponding to an application, have a weight of 5, and all the edges, representing data exchange between applications, have a weight of 1. For the sake of clarity, in this example we are going to conceive each partition load as the summation of all its vertices' load. In the left-most graph a cut divides it in two partitions, the first one with 6 nodes and a total weight of 30, and the second one with 2 nodes and total weight of 10. The total edge cut is 3. Using the formula in equation (18) we obtain a total cost of 0.0133. To reduce this cost, in the middle graph we draw a new cut that creates two partitions and whose total edge cut is 1. The first partition's load is 35, whereas the second one is 5. This time the cost function favors partitions a bit less balanced than the previous one, but significantly reduces the data exchange from 3 to 1. Finally, in the right-most graph we choose a new cut that divides the graph in a perfectly balanced 2 partitions whose weights are 20. The new edge cut is slightly higher than the previous one, this is 2. It is not difficult to see that this is the best cut among the 3 options. As expected, the cost function produces its lowest

value with the last cut, this is 0.0075. Clearly, the cost function favors both more balanced partitions and less inter-partition dependencies, or data exchanges (see Sections 3.2.A and 3.2.B).

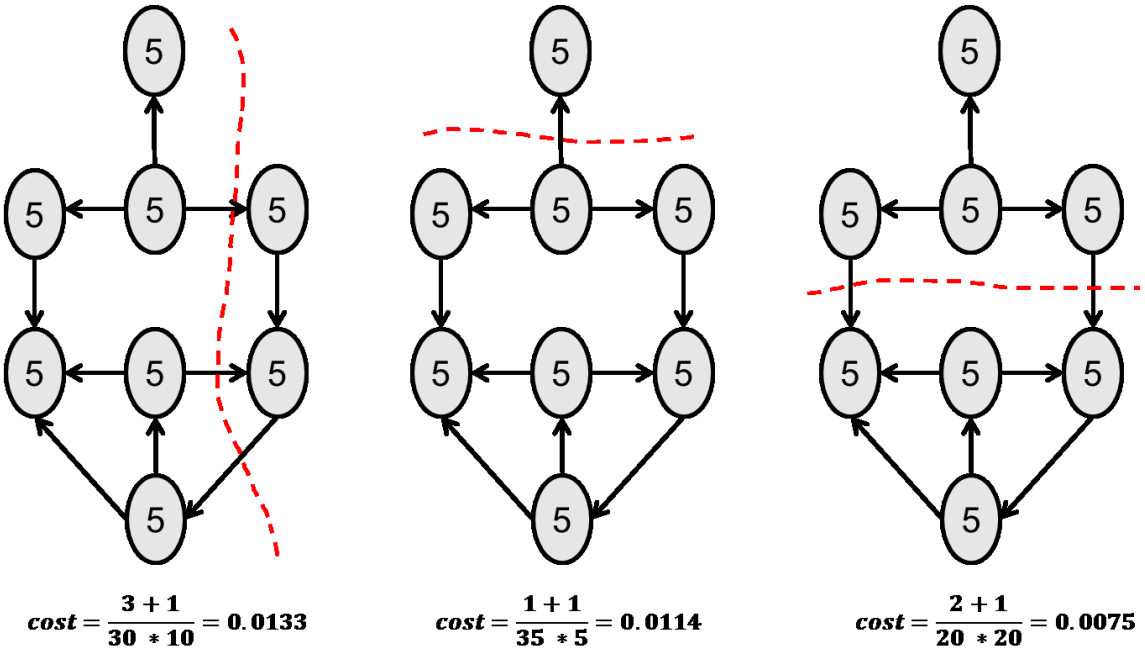

Figure 8. Cost function example

Finding a global solution to the optimization problem addressed in this step is unfeasible. Instead, we have defined an extra heuristic that is very simple, converges fast, and attempts to find optimal transfers of applications between scheduler trees. We first need to calculate the cost of the initial partition built by the previous two steps. Next we enter in an iterative, k-way partitioning process. At each iteration we compute all possible transfers and try to find the one that reduces the cost the most. A transfer simply means moving an application from its current scheduler tree to a different scheduler tree. A transfer is feasible if there is capacity in the target scheduler tree to accommodate the application. In such case, we calculate the cost of the new partitions in the imaginary case

that this transfer were executed. Next, we find the transfer that minimizes the cost the most. If this imaginary new cost is actually lower than the current partitions cost, we execute the transfer permanently and update the new partitions cost. Additionally, we remove the transferred application from further consideration. Also note that in this step we only consider non-critical applications for transfers. Involving critical applications would require considering the RESTRICTED_LOOP. Additionally, critical applications are unlikely to produce feasible transfers since we must move them all, or none of them. Yet, should the transfer be feasible, there is no guarantee it will minimize the cost. This step finishes when there is no more applications to be considered, or when the cost cannot longer be minimized. The pseudo code for this step is shown below in Listings 6, 7, 8 and 9.

```
function balancingStep():
1:  cost ← calculate costFunction()
2:  apps← find non-critical applications in APPS
3:  do:
4:      transfers←compute 3-tuples (app, sourceSchedulerTree,
    targetSchedulerTree) for all possible transfers in apps
5:      iteration_costs←create empty list of duple(transfer,cost)
6:      for each t in transfers do:
7:          if t is feasible then:
8:              c←calculate costFunction() if t were executed
9:              add duple c,t to iteration_cost
10:         end if
11:     end for
12:     minTransfer←find transfer with min cost in iteration_cost
13:     if minTransfer is not NULL and minTransfer.cost<cost then:
```

```
14:         executeTransfer(minTransfer)
15:         remove minTransfer.transfer.app from apps
16:         cost←minTransfer.cost
17:     end if
18: while (cost is minimized and apps is not empty)
```

Listing 6. Pseudo-Code for the Balancing Step

```
function costFunction():
1:  edgeWeight←1
2:  loadProduct←1
3:  for each tree in SCHEDULERS do:
4:      edgeWeight← edgeWeight + schedulerTreeOutEdgesWeight(tree):
5:      loadProduct ← loadProduct * schedulerTreeLoad(tree)
6:  end for
7:  return edgeWeight / loadProduct
```

Listing 7. Pseudo-Code for the K-way Ratio-Cut Cost Function

```
function schedulerTreeLoad(schedulerTree):
1:  load←1
2:  for each vertex in VERTICES IN (schedulerTree) do:
3:      for each app in vertex do:
4:          level← targetDepthLevel(app.ExecRate)
5:          appLoad←app.ExecTime*2^(SCHEDULER_TREE_LEVELS–
    level)
6:          load←load + appLoad
7:      end for
8:  end for
9:  return load
```

Listing 8. Pseudo-Code for calculating the total load of a given scheduler tree

```
function schedulerTreeOutEdgesWeight(schedulerTree):
1:  outWeight←0
2:  for each vertex in VERTICES IN (schedulerTree) do:
3:      for each app in vertex do:
4:          edges←DEPENDENCIES.outEdges(app)
5:          for each e in edges do:
6:              if e.targetApp is not registered in schedulerTree then:
7:                  outWeight←outWeight + e.weight
8:              end if
9:          end for
10:     end for
11: end for
12: return outWeight
```

Listing 9. Pseudo-Code for calculating the outgoing edges weight of a given scheduler tree

Figure 9 depicts a candidate configuration built after running the first two steps of this algorithm. The example shows 7 applications arranged into 2 partitions. For simplicity, let us assume that all the data exchange between applications is equal to 1 and all applications are non-critical. As it can be seen in any of the 2 partitions, some nodes hold 2 or more applications, whereas others hold only 1. This is a perfectly valid scenario in which the first two steps tried their best to build scheduler trees whose execution paths' load are balanced. The tree 1 has 4 execution paths, each of them holding a load of either 5 or 6. Likewise, the tree 2 has 4 execution paths, each of them holding a load of 2. However, the load between the scheduler trees is clearly not balanced. The inter-partition dependencies is pretty fair, and the only way to reduce it would be moving App 6 and/or App 7 to Tree 1, which in turn will create more load imbalanced between the partitions. One of the key concepts introduced by this balancing step is *possible transfers*. In this

example all applications are transferable as they are all non-critical. Table 1 summarizes all possible transfers at the first iteration, and the hypothetical new cost if the transfers were to be executed. The cost of the initial partitions is 0.0170. It should not be surprising that transfering either App 6 or App 7 to Tree 1 increases this cost to 0.0192. Likewise, transfering App 2 to Tree 2 is a poor decision that would increase the cost to 0.0231. It slightly improves the load balance at the expense of duplicating the inter-partition dependencies. It is easy to see that the best options would be transfering either App 4 or App 5 to Tree 2. They do not augment the data exchange between the partitions and result in more balanced trees. However, choosing App 5 over App 4 is a better decision as it produces even more balanced trees. As a result, at the first iteration of this step the algorithm transfers App 5 to Tree 2 permanently and it is not longer considered for further transfers in subsequent iterations. Figure 10 shows how the partitions look after completing the first iteration. The new cost to be minimized is 0.0133.

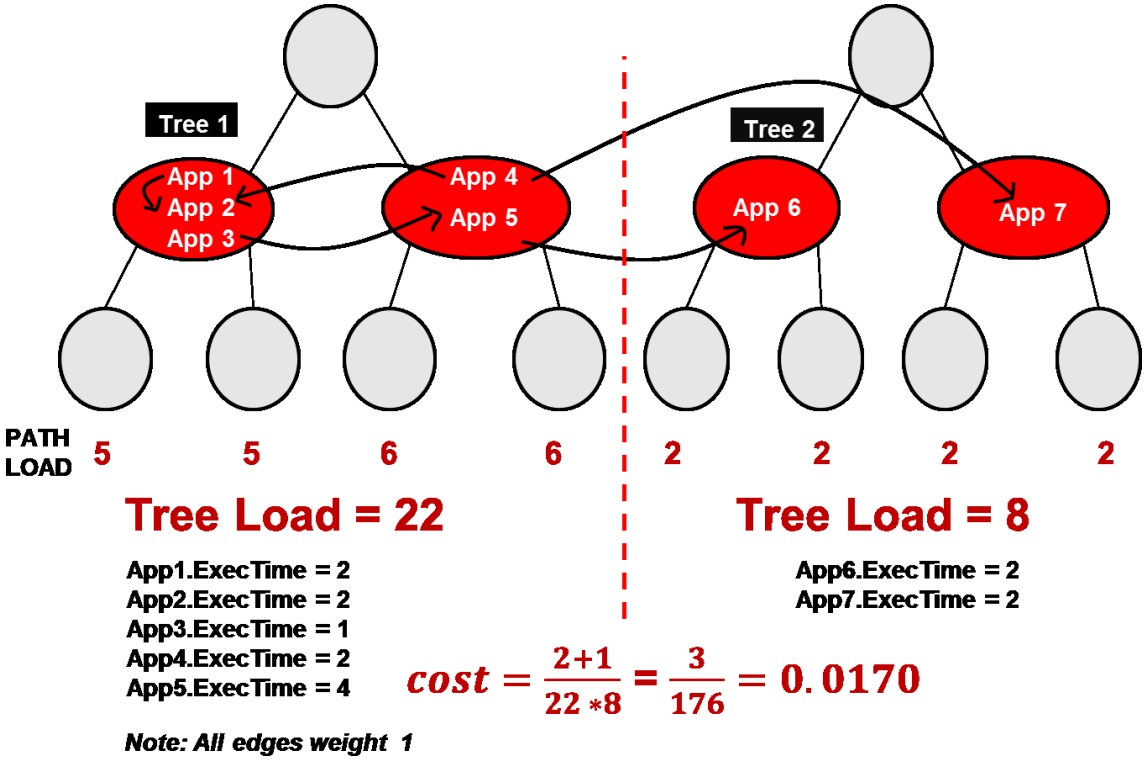

Figure 9. Best transfer example (1/2)

Application	From	To	Hypothetical new cost
App1	Tree1	Tree2	$cost = \frac{3+1}{18 * 12} = 0.0185$
App2	Tree1	Tree2	$cost = \frac{4+1}{18 * 12} = 0.0231$
App3	Tree1	Tree2	$cost = \frac{3+1}{20 * 10} = 0.0200$
App4	Tree1	Tree2	$cost = \frac{2+1}{18 * 12} = 0.0138$
App5	Tree1	Tree2	$cost = \frac{2+1}{14 * 16} = 0.0133$
App6	Tree2	Tree1	$cost = \frac{1+1}{26 * 4} = 0.0192$
App7	Tree2	Tree1	$cost = \frac{1+1}{26 * 4} = 0.0192$

Table 1. Best Transfer Example

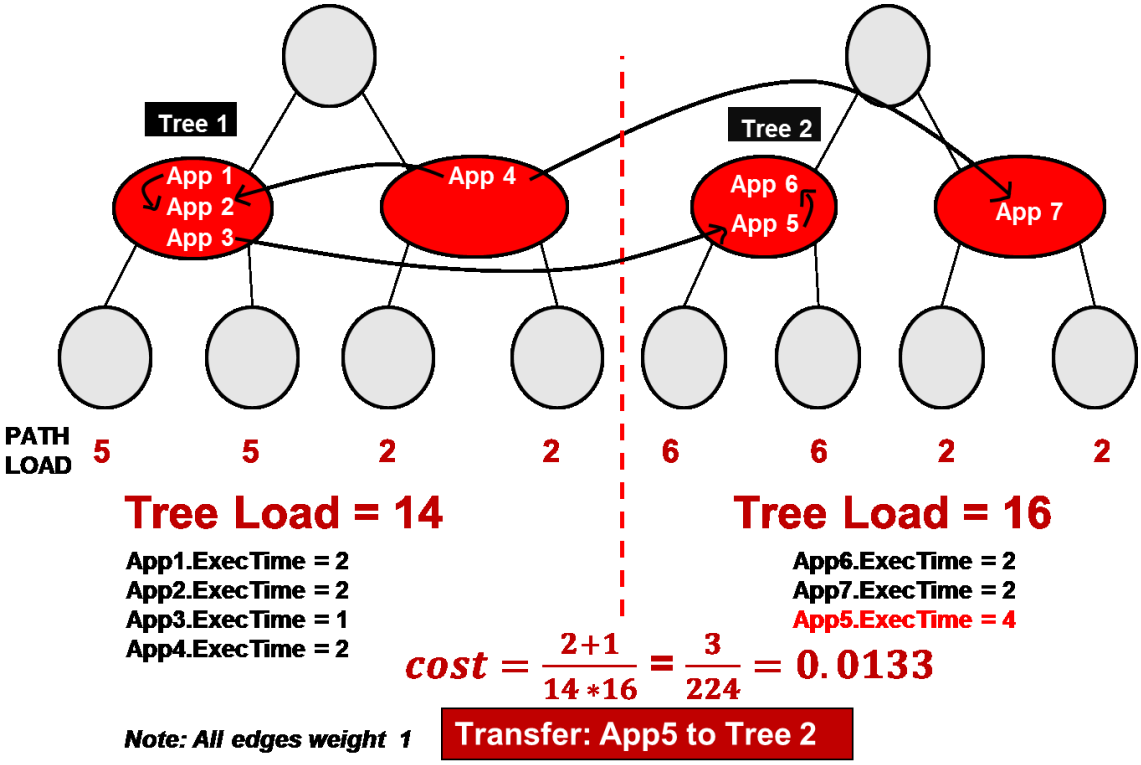

Figure 10. Best Transfer Example (2/2)

Chapter 4 Evaluation

To evaluate the effectiveness of our approach we have conducted a case study at CAE. We have asked an integration specialist to build the configuration files for a specific flight simulator model using different inputs (e.g., number of schedulers, applications, etc.). We have run our algorithm to build configuration files for the same model using the same input.

Next, we have run the simulation software using both the human-based and algorithm-based configuration files and compared the results in terms of performance. Performance is measured using critical and non-critical overruns metrics (see Section 2.1.1). As we will see later, the outcome of this case study is encouraging. In some cases we were able to produce configuration files that outperformed those produced by an integration specialist. In some other cases we produced configuration files that were almost as good as those built by the technician.

4.1 Flight Simulator Model (applications, dependencies and restricted-loop)

At CAE, engineers use virtual labs specifically designed to emulate the behaviour of real mechanical components. As in a normal scenario, there is a middleware used to transport the information between the mechanical part and the simulation unit. As far as the simulation unit is concerned, the simulation environment is exactly the same as the real one. For this experimental study, we have used one virtual lab.

With regard to the simulation software, we use a generic aircraft simulation platform, comprised of approximately 50 applications and their dependencies (see Sections 3.3.1 and 3.3.2). In the rest of this work we will simply refer to this platform as GAS. Each application simulates a specific component of the airplane, or its environment. For instance: air, weather, radar, flight controls, aerodynamic, etc. As the reader may sense, some of the applications in the GAS Platform form the so called restricted-loop (see Section 3.3.4). A list of all applications used in this study, their dependencies and the formal definition of the restricted-loop of the GAS Platform is omitted in this work since it is extremely sensitive information.

4.2 Configuration Scenarios (Human-based vs. Algorithm-based configurations)

To carry out this experimental study we have used exactly 2 configuration scenarios:

a. 2-Scheduler Configuration: The 50 applications comprising the GAS Platform are grouped into 2 scheduler trees.

b. 3-Scheduler Configuration: The 50 applications comprising the GAS Platform are grouped into 3 scheduler trees.

The abovementioned inherently specifies the number of required scheduler trees (see section 3.3.5). Note that both configuration scenarios are constrained by the following:

- The scheduler tree depth level is 5 (see section 3.3.6)
- The execution path real time budged is 60 Hertz (see section 3.3.7)

- The execution path virtual time budget is 0.30 (see section 3.3.8)

These configuration scenarios were not arbitrarily designed. We used previous working configurations and met with CAE engineers to properly define them. Indeed, these configuration scenarios resemble the common ones used by CAE to setup the flight simulators they commercialize. Lastly, we have asked an integration specialist to build both configuration scenarios based on the best of his knowledge. We call them *Human-Based* configurations. Likewise, we use our algorithm to build the two of them. We call them *Algorithm-Based* configurations.

4.3 Simulation Scenarios

For this case study, we have defined two simulation scenarios: a) Aircraft on ground; b) Aircraft in air. As a matter of fact, these scenarios were highly suggested by CAE Engineers. According to them, simply positioning the aircraft on ground and in air is enough to execute most of the source code execution paths of all the applications involved in the simulation. Due to this reason, we left out most specific simulation scenarios such as an aircraft crashing, landing, increasing or decreasing altitude, etc.

4.3.1 Aircraft on Ground

The aircraft is in take-off position. The engine is started and all applications are running. This scenario allows simulating the very exact moment when the aircraft is just about to start taking off.

4.3.2 Aircraft in Air

The aircraft is positioned at an altitude of approximately 10.000 feet. All flight related applications are running. The flight is unfreeze, which means that the aircraft is indeed moving horizontally in the air. The altitude is freeze. This means that the aircraft cannot move vertically in the air. As there is no pilot, we need to freeze the altitude, otherwise the aircraft would crash. Note that many applications are also simulating the environment in which the aircraft is. Considering the latter, we placed a heavy storm right on top of the aircraft and a turbulence of around 64%.

4.4 Experiment definitions and data collection process

To conduct this experimental study, we execute every configuration scenario (human-based and algorithm-based) against each simulation scenario. To make our results more solid and to avoid false positives, each combination of configuration scenario / simulation scenario is run exactly tree times. In overall, we have run 24 experiments, this is:

- 4 configuration scenarios: 2 human-based and 2 algorithm-based.
- Multiplied by 2 simulation scenarios.
- Multiplied by 3 runs each

A table depicting all possible combinations is shown below:

Id	Configuration	Simulation	Times
C21	2-Scheduler-Human	On Ground	3
C22	2- Scheduler -Human	In Air	3
C23	2- Scheduler -Algorithm	On Ground	3

C24	2- Scheduler -Algorithm	In Air	3
C31	3- Scheduler -Human	On Ground	3
C32	3- Scheduler -Human	In Air	3
C33	3- Scheduler -Algorithm	On Ground	3
C34	3- Scheduler -Algorithm	In Air	3

Table 2. List of Experiments

Each scheduler is in charge of producing statistical information in the end of every scheduler cycle for debugging and testing purposes. CAE Engineers use this information to find problems and to improve the performance of the simulation. For this case study we are only concerned about critical and non-critical overruns (see section 2.1.1). At each scheduler iteration, an updated version of the statistics overrides the previous one. Note that for this study each scheduler of all configuration scenarios is running at 60Hz (see section 4.2). This means that any scheduler produces a new set of statistics approximately every 16.7ms. Trying to capture a *snapshot* of these statistics every 16.7ms is not only impractical, but would also significantly degrade the performance of the simulation. Taking only 1 snapshot in the end of the simulation would allow us to see the final performance results, but we would not be able to see the evolution of these statistics along the simulation. Considering this, we take a snapshot of the statistics approximately every 25sec; this is, every 1500 scheduler cycles:

SNAPSHOT_TIME = 16,667ms * 1500 iteration = 25000.5 ms/snapshot

We run each experiment for approximately 31mins. This is:

EXPERIMENT_TIME = 31mins = 31 * 60sec * 1000ms = 1860000ms

Based on the previous two definitions it is easy to see that the total number of required statistical data snapshot per scheduler is 75:

TOTAL_SNAPSHOTS = EXPERIMENT_TIME / SNAPSHOT_TIME

TOTAL_SNAPSHOTS = 1860000ms / 25000.5 ms/snapshot

TOTAL_SNAPSHOTS = 74.39 snapshot

Note that some configurations run with 2 schedulers, while others run with 3 schedulers. The number of statistical snapshots is proportional to the number of schedulers for a given configuration. To illustrate, for a given 2-scheduler configuration we collect 150 snapshots, this is, 75 for each scheduler.

4.5 Data results & Analysis

To better understand the results, we carry out this analysis by comparing the outcome of human-based configurations against algorithm-based ones, both in a 2-Scheduler and a 3-Scheduler configuration (see section 4.2). As it was previously pointed out, the main goal of this study is to evaluate the performance of human-based and algorithm-based configurations in terms of critical and non-critical overruns.

4.5.1 2- Scheduler Configuration, On Ground

Figure 11 shows the average critical and non-critical overruns obtained in the first scheduler after executing the 3 trials, both for human-based and algorithm-based configurations, with the aircraft on ground and using a 2-Scheduler configuration. Note that this first scheduler is used to register all critical applications. Clearly the algorithm-based configuration (in red) outperformed the human-based one (in blue). In average, the

algorithm-based configuration produced 9.3 and 11 critical and non-critical overruns respectively. On the other hand, the human-based one caused 15 and 22 critical and non-critical overruns respectively.

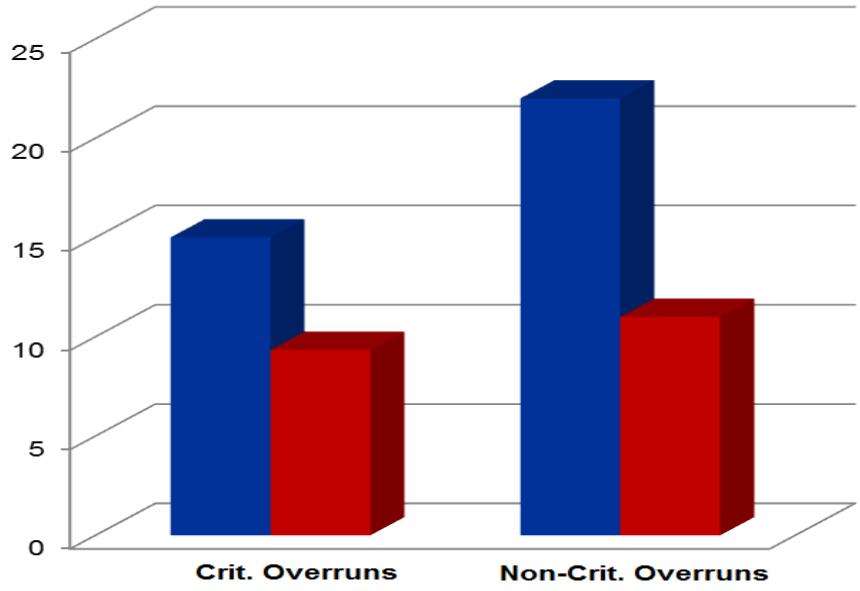

Figure 11. Exp: 2-Scheduler (1/2), On Ground. Human-based in Blue; Algorithm-based in Red

Figure 12 shows the average critical and non-critical overruns obtained in the second scheduler after executing the 3 trials, both for human-based and algorithm-based configurations, with the aircraft on ground and using a 2-Scheduler configuration. Once again, the algorithm-based configuration (in red) surpassed the human-based one (in blue) when it comes to critical overruns. In average, the algorithm-based configuration produced 4.6 critical overruns, while the human-based one caused 8.3. However, when it comes to non-critical overruns both configurations behaved similarly. The algorithm-based one provoked 9.6 non-critical overruns, while the human-based one generated 9.3.

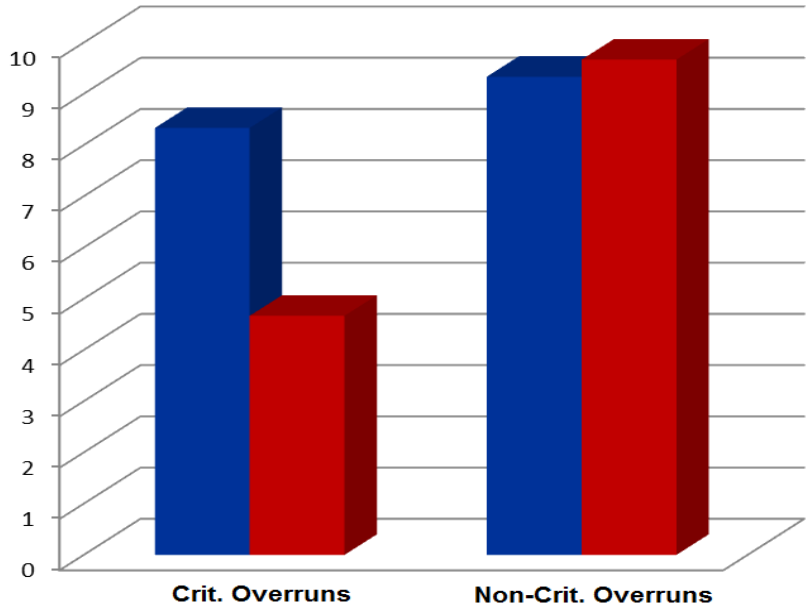

Figure 12. Exp: 2-Scheduler (2/2), On Ground. Human-based in Blue; Algorithm-based in Red.

The algorithm was capable of producing a better distribution of the applications between the 2 schedulers, and this is likely the major reason why its configuration produced better results than that of the human. More specifically, the algorithm added a total load of 51780.00μs (microseconds) to the first scheduler and 51600.01μs to the second one. Many non-critical applications that were not highly dependent on the critical ones were placed in the second scheduler. The algorithm produced more balanced partitions when compare with those of the human-based configuration. More exactly, the human added 70715.46μs to the first scheduler and 32664.64μs to the second one. As the reader can observe, the first scheduler of the human-based configuration was far more loaded than the algorithm-based one, therefore producing worse results. However, the second scheduler of the human-based configuration was considerably less loaded than the algorithm-based one, but this was not enough to produce significant better results as both configurations performed similarly. The key to understand this behaviour is not how full

one scheduler is, but how much room there is available to face non-uniform application execution times. For example, a given application called AircraftDynamics might be expected to execute in 800μs, but in very few scheduler cycles this execution time might be way higher, say 1200μs. If such situation happens to many applications in the same scheduler cycle, this will certainly cause one or more critical or non-critical overruns. The reason why both human-based and algorithm-based configurations performed equally in the second scheduler is likely due to the fact that having a scheduler up to 20% of its capacity is good enough to handle most of the non-uniform execution times. Note that the real time budget per execution path is 60 Hertz (see section 4.2), the theoretically capacity of any scheduler in all the experiments is 16666.67μs per execution path multiplied by 16 possible execution paths, this is 266666.67μs. Based on this, the second scheduler of the human-based configuration is at 12.24% of its capacity while the algorithm-based one is at 19.35%.

In reality, distributing the load among partitions is necessary but not sufficient. It is still indispensable to look at the load distribution among execution paths for every scheduler tree. In our experiments all scheduler trees count with 16 execution paths. If 15 execution paths were to be absolutely empty and only one carried with the entire load, this would certainly be a problematic scheduler tree causing thousands of critical and non-critical overruns in short time. Table 3 shows the average execution path time in microseconds for every execution path in both human-based and algorithm-based configuration schedulers. We have executed any experiment 3 times and collected around 111600 (this is approximately the number of scheduler cycles in 31mins execution time for a scheduler running at 60Hertz) statistical samples per execution path, scheduler and experiment trial.

Exec Path	Human Sched 1	Algorithm Sched 1	Human Sched 2	Algorithm Sched 2
1	3407.33	2542.00	1379.67	2268.33
2	3284.00	2453.00	1379.67	2454.67
3	3418.00	2452.83	1379.67	2268.33
4	3283.33	2453.33	1379.67	2306.00
5	3585.33	2438.67	1379.67	1837.33
6	3219.33	2438.33	1379.67	2070.67
7	3041.00	2439.00	1379.67	1837.33
8	3041.33	2439.00	1379.67	1879.33
9	3107.67	2451.67	1379.67	2709.33
10	3040.67	2451.67	1379.67	3171.00
11	3839.33	2451.33	1379.67	2709.33
12	3772.67	2451.00	1379.67	2751.00
13	3839.33	2611.33	1379.67	2336.67
14	3772.67	2611.33	1379.67	2628.00
15	4017.00	2611.33	1379.67	2336.67
16	3272.33	2611.33	1379.67	2384.67
AVG	3419.44	2494.20	1379.67	2371.79
STD	0320.85	0071.53	0000.00	0354.94

Table 3. Avg. execution path time in µs of schedulers in the 2-Schedulers Configuration, On Ground, experiments

As it can be seen, in the first scheduler of the algorithm-based configuration the average execution path time is not only significantly lower than that of the human-based one, but also the load is better evenly distributed among all execution paths, this is, a 71.53µs standard deviation. This is another reason why the first scheduler of the algorithm-based configuration performed better than the human-based one. As we can also see, in the second scheduler of the human-based configuration the load was perfectly distributed among all execution paths. The reason behind this is that the integration specialist placed all the applications in the scheduler tree's root node, leaving all the remaining nodes empty. However, in average the integration specialist used only 8.27% of the capacity of all execution paths. On the other hand, our algorithm placed more applications in the second scheduler. In average, it filled the execution paths capacity up to 14.23%, which

produced better results in terms of critical overruns and similar ones in terms of non-critical overruns. The theoretically execution path capacity of any execution path is 16666.67µs (based on 60Hertz).

4.5.2 2- Scheduler Configuration, In Air

Figure 13 shows the average critical and non-critical overruns obtained in the first scheduler after executing the 3 trials, both for human-based and algorithm-based configurations, with the aircraft in air and using a 2-Scheduler configuration. Once again, algorithm-based configuration (in red) outperformed the human-based one (in blue). In average, the algorithm-based configuration produced 12 and 13 critical and non-critical overruns respectively. On the other hand, the human-based one caused 20.67 and 29.67 critical and non-critical overruns respectively.

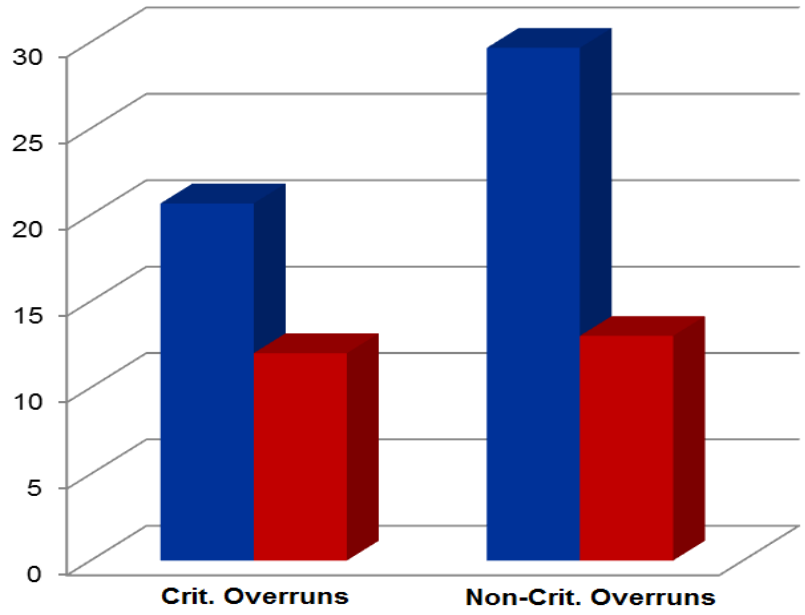

Figure 13. Exp: 2-Scheduler (1/2), In Air. Human-based in Blue; Algorithm-based in Red

Likewise, Figure 14 shows the results obtained in the second scheduler for the same experiments. As for the second scheduler in the previous subsection, the algorithm-based configuration (in red) surpassed the human-based one (in blue) when it comes to critical overruns. In average, the algorithm-based configuration produced 6 critical overruns, while the human-based one caused 12.67. Additionally, both set of configurations produced in average equal number of non-critical overruns, this is 12.67. The reasons why the algorithm-based configuration clearly outperformed the human-based configurations are exactly the same reasons pointed out in the previous subsection. These are, the total load was better and evenly distributed among the partitions, and even more, very well distributed among all the execution paths of the two scheduler trees.

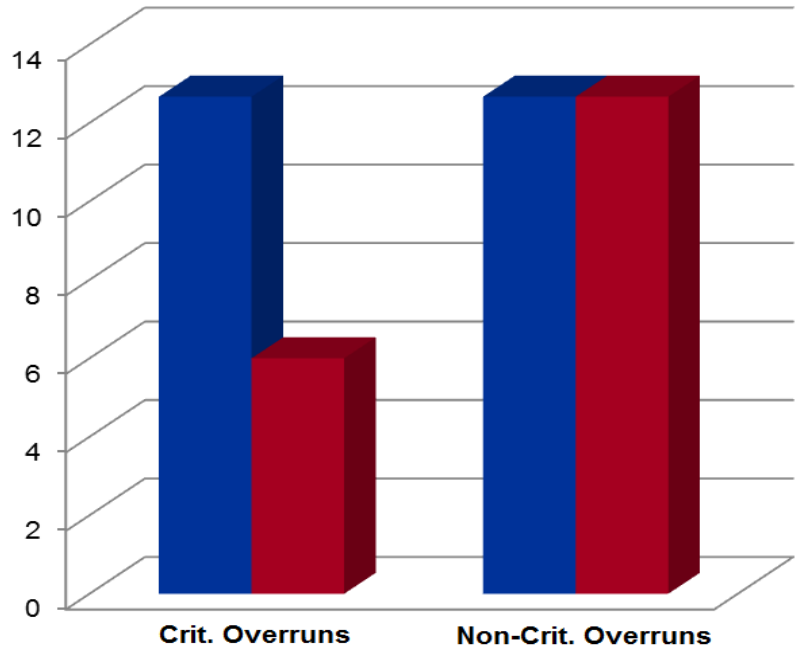

Figure 14. Exp: 2-Scheduler (2/2), In Air. Human-based in Blue; Algorithm-based in Red

Table 4 shows the average execution path time in microseconds for every execution path in both human-based and algorithm-based configuration schedulers. As for the previous

experiments, we have executed any experiment 3 times and collected around 111600 statistical samples per execution path, scheduler and experiment trial.

Exec Path	Human Sched 1	Algorithm Sched 1	Human Sched 2	Algorithm Sched 2
1	4694.33	3619.67	1962.33	2455.00
2	4549.67	3532.33	1962.33	2687.33
3	4691.33	3532.00	1962.33	2455.00
4	4549.33	3531.67	1962.33	2498.67
5	5337.67	3501.33	1962.33	2006.67
6	4717.33	3501.33	1962.33	2234.67
7	4862.33	3501.00	1962.33	2006.67
8	4717.33	3501.33	1962.33	2045.00
9	4309.67	3801.67	1962.33	2920.67
10	4309.67	3802.67	1962.33	3366.33
11	4380.33	3801.67	1962.33	2920.67
12	4309.33	3802.67	1962.33	2983.00
13	5236.67	3677.33	1962.33	2643.67
14	5162.67	3677.33	1962.33	2972.33
15	5444.33	3677.00	1962.33	2643.67
16	5162.33	3677.33	1962.33	2703.33
AVG	4777.15	3633.65	1962.33	2596.42
STD	0371.91	0069.23	0000.00	0113.48

Table 4. Avg. execution path time in µs of schedulers in the 2-Schedulers Configuration, In Air, experiments

4.5.3 3-Scheduler Configuration, On Ground

The following three charts show the average critical and non-critical overruns generated in the three schedulers after executing the 3 trials, both for human-based (in blue) and algorithm-based (in red) configurations with the aircraft on ground. Note that the first scheduler is used to register all critical applications. With regard to the first, critical, scheduler, once again the algorithm-based configuration outperformed the human-based one (see Figure 15). In average, the algorithm-based configuration produced 16 and 16.3 critical and non-critical overruns respectively. On the other hand, the human-based one

caused 21.67 and 26 critical and non-critical overruns respectively. These results are not surprising and the most likely reasons for this behaviour are the same ones pointed out in the previous subsections: when comparing to the human, the algorithm was able to assign less load to the first scheduler and to better distribute the load among all the scheduler execution paths.

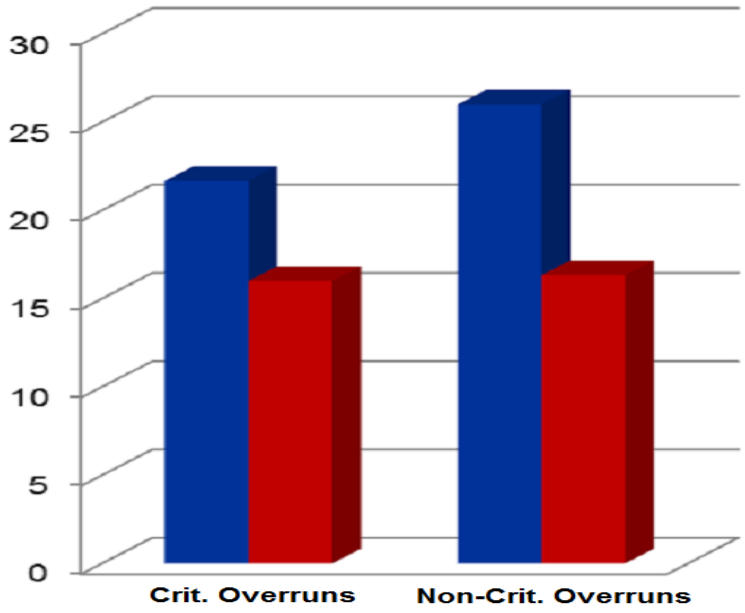

Figure 15. Exp: 3-Scheduler (1/3), On Ground. Human-based in Blue; Algorithm-based in Red

An interesting outcome of these experiments is observed in the second and third schedulers. In both of them, the human-based configuration clearly performed much better than its algorithm-based counterpart. In the second scheduler, the human-based configuration generated in average 8 and 8.3 critical and non-critical overruns respectively, while the algorithm-based one produced 12.3 and 20.67 (see Figure 16). Similarly, in the third scheduler, the human-based configuration caused in average 2.67 and 9 critical and non-critical overruns, while the algorithm-based one generated 4 and

20.3 (see Figure 17). Special attention must be paid to the third scheduler, as this was bound to a shared CPU. Logically, the integration specialist posed almost no load on it, registering only one application in it, the flight management guidance system, which in average needed only 77.67µs to execute in all execution paths. The algorithm is not aware of this fact, and distributes the load among the schedulers as if they were all equally free. In average, the third scheduler's execution paths needed 1059.88µs to execute (see table 5).

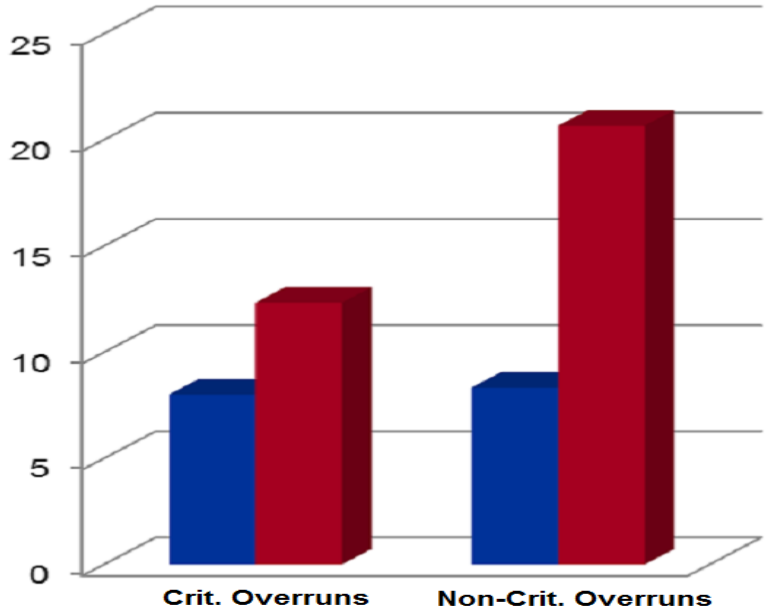

Figure 16. Exp: 3-Scheduler (2/3), On Ground. Human-based in Blue; Algorithm-based in Red

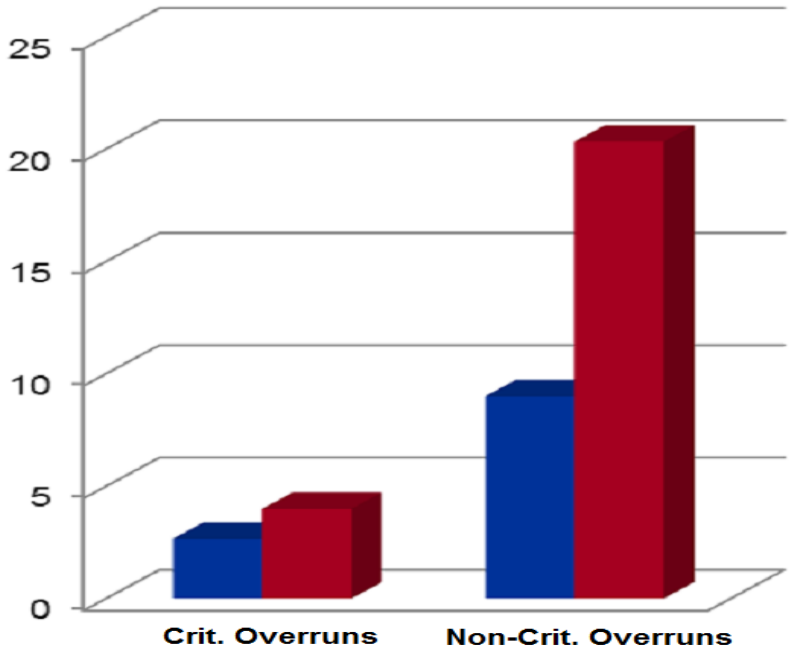

Figure 17. Exp: 3-Scheduler (3/3), On Ground. Human-based in Blue; Algorithm-based in Red

Table 5 shows the average execution path time in microseconds for every execution path in both human-based and algorithm-based configuration schedulers. As for the previous experiments, we have executed any experiment 3 times and collected around 111600 statistical samples per execution path, scheduler and experiment trial.

Exec Path	Human Sched 1	Algorithm Sched 1	Human Sched 2	Algorithm Sched 2	Human Sched 3	Algorithm Sched 3
1	3458.00	2325.67	1341.00	1351.67	0077.67	0979.33
2	3333.67	2326.00	1341.00	1351.67	0077.67	1204.67
3	3472.33	2326.33	1341.00	1351.67	0077.67	0979.33
4	3333.33	2326.00	1341.00	1429.33	0077.67	1017.00
5	3948.00	2326.00	1341.00	1567.33	0077.67	0519.00
6	3270.00	2326.00	1341.00	1567.33	0077.67	0743.00
7	3416.67	2326.67	1341.00	1842.33	0077.67	0519.00
8	3269.67	2327.00	1341.00	1842.33	0077.67	0553.33
9	3110.33	2451.67	1341.00	1465.00	0077.67	1382.00
10	3110.00	2451.67	1341.00	1465.00	0077.67	1830.33
11	3177.00	2451.33	1341.00	1465.00	0077.67	1382.00
12	3110.00	2451.67	1341.00	1465.00	0077.67	1417.67
13	3909.00	2395.67	1341.00	1504.33	0077.67	1031.33

14	3840.00	2395.67	1341.00	1504.33	0077.67	1287.33
15	4088.33	2395.33	1341.00	1504.33	0077.67	1031.33
16	3840.33	2395.67	1341.00	1504.33	0077.67	1081.33
AVG	3480.42	2374.90	1341.00	1511.31	0077.67	1059.88
STD	0323.13	0033.48	0000.00	0071.96	0000.00	0092.12

Table 5. Avg. execution path time in µs of schedulers in the 3-Schedulers Configuration, On Ground, experiments

4.5.4 3-Scheduler Configuration, In Air

The following thee charts show the average critical and non-critical overruns generated in the three schedulers after executing the 3 trials, both for human-based (in blue) and algorithm-based (in red) configurations with the aircraft in air. As in the previous 3-scheduler experiments, the human-based configuration clearly outperformed the algorithm-based one. One of the most interesting results is produced in the first scheduler (see Figure 18), in which the human-based configuration caused in average 14 critical overruns, while the algorithm-based one produced 19.3. In terms of non-critical overruns, both configurations performed similarly producing in average 21 per experiment. This was the only case in which the human was capable of producing better results in the first, critical, scheduler. The average execution time of all execution paths for the algorithm-based and human-based configurations were 3509.08µs and 4838.46µs respectively. Even more, the algorithm was capable of better distributing the load among all execution paths. In average, the execution time standard deviations of all execution paths for the algorithm-based and human-based configurations were 64.18µs and 369.50µs respectively. Yet, the human was able of producing better results. An in-depth analysis shows an irregular behaviour in the algorithm experiment first trial. In approximately 30sec, from the snapshot 53 to the 54, the total number of critical overruns went from 17 to 30, an increase of 76.47%. At the end, this trial alone caused 36 critical overruns,

while the others two generated 8 and 14. We believed this abnormal behaviour might have been caused by non-simulation related components or factors. For example, the operating system scheduler mechanism could have taken the CPU from the simulation process for an unusual, long period of time.

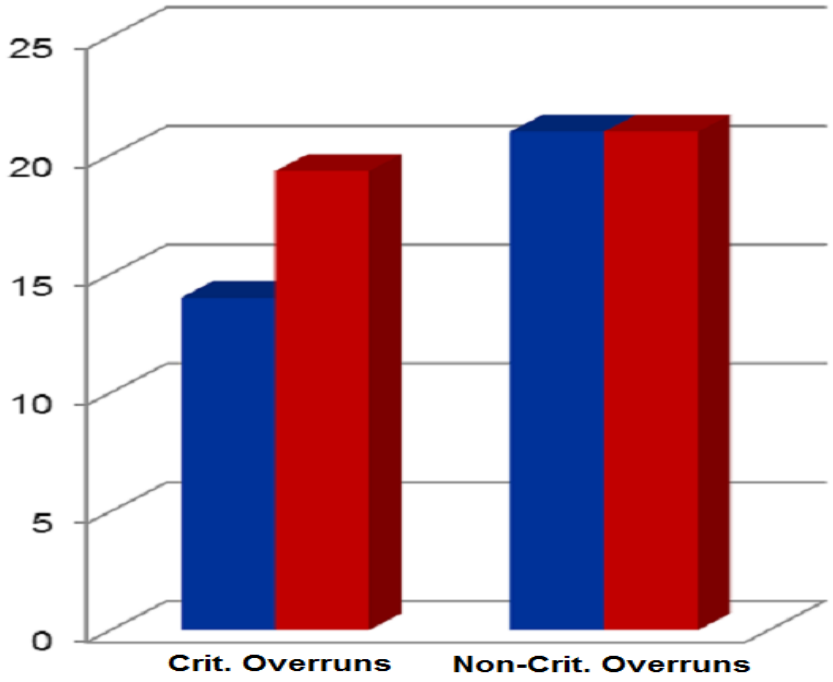

Figure 18. Exp: 3-Scheduler (1/3), In Air. Human-based in Blue; Algorithm-based in Red

The second scheduler also shows an interesting result (see Figure 19). In average, the algorithm-based configuration caused 8.7 and 21 critical and non-critical overruns respectively, while the human-based one generated 12.67 and 13.67. The average execution time of all execution paths for the algorithm and human-based configurations were 1655.27µs and 1393.33µs respectively. The human placed the entire load in the critical node, which means that the execution time standard deviation of all execution paths equaled to 0µs, while the algorithm-based one equaled to 157.33µs. Even though

the human-based configuration caused less non-critical overruns, the algorithm-based one performed much better as it produced less critical overruns. The final and third scheduler shows results not far different from those of the previous experiments in section 4.5.3 (see Figure 20). The human placed almost no load in it, probably knowing in advance that the CPU was shared with another process. The algorithm is not aware of this fact and tried to balance the load among all the schedulers. This resulted in a far more loaded third scheduler when compared with the human-based one. This is likely the reason why the latter produced better results. In average, the algorithm-based configuration caused 11 and 22.67 critical and non-critical overruns, while the human-based one generated 7.67 and 12.67.

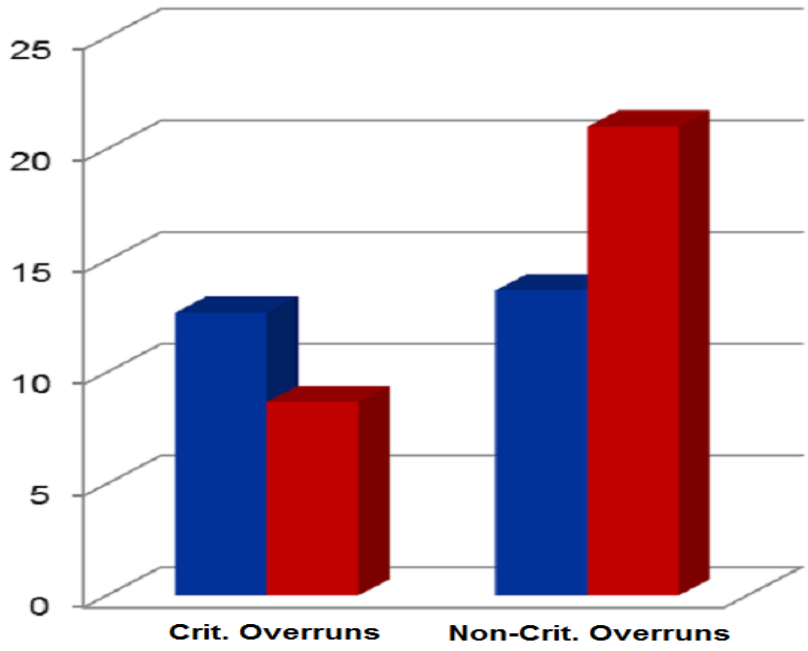

Figure 19. Exp: 3-Scheduler (2/3), In Air. Human-based in Blue; Algorithm-based in Red

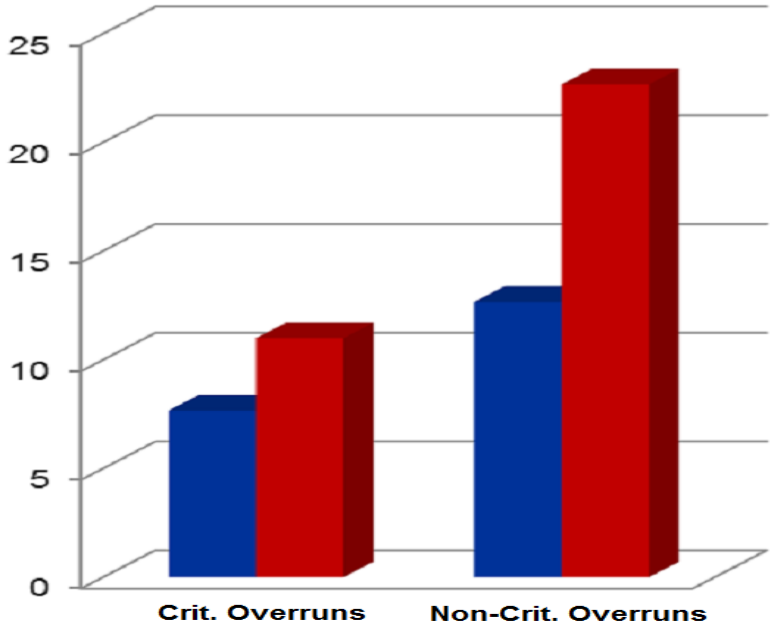

Figure 20. Exp: 3-Scheduler (3/3), In Air. Human-based in Blue; Algorithm-based in Red

Table 6 shows the average execution path time in microseconds for every execution path in both human-based and algorithm-based configuration schedulers. As for the previous experiments, we have executed any experiment 3 times and collected around 111600 statistical samples per execution path, scheduler and experiment trial.

Exec Path	Human Sched 1	Algorithm Sched 1	Human Sched 2	Algorithm Sched 2	Human Sched 3	Algorithm Sched 3
1	4752.67	3453.00	1393.33	1683.67	0583.00	1631.33
2	4604.67	3453.00	1393.33	1683.67	0583.00	1879.00
3	4745.00	3452.67	1393.33	1683.67	0583.00	1631.33
4	4605.00	3452.67	1393.33	1774.67	0583.00	1675.33
5	5398.00	3448.33	1393.33	1628.67	0583.00	1177.00
6	4777.33	3448.00	1393.33	1628.67	0583.00	1404.67
7	4926.33	3448.00	1393.33	2026.00	0583.00	1177.00
8	4776.67	3448.33	1393.33	2026.00	0583.00	1214.33
9	4379.00	3604.67	1393.33	1523.00	0583.00	2050.33
10	4379.00	3604.67	1393.33	1523.00	0583.00	2505.33
11	4447.33	3604.67	1393.33	1523.00	0583.00	2050.33
12	4379.00	3604.67	1393.33	1523.00	0583.00	2112.33
13	5297.67	3530.67	1393.33	1564.33	0583.00	1805.67
14	5224.67	3530.67	1393.33	1564.33	0583.00	2072.33

15	5498.33	3530.67	1393.33	1564.33	0583.00	1805.67
16	5224.67	3530.67	1393.33	1564.33	0583.00	1857.33
AVG	4838.46	3509.08	1393.33	1655.27	0583.00	1753.08
STD	0369.50	0064.18	0000.00	0157.32	0000.00	0363.74

Table 6. Avg. execution path time in µs of schedulers in the 3-Schedulers Configuration, In Air, experiments

Chapter 5 Conclusion

Generating configuration scenarios for the real-time aircraft simulation systems used at CAE, such as flight simulators, is a complex process. A configuration refers to the distribution of the execution of all applications comprising these systems among different processors. To do this, applications are arranged into binary trees, also called *scheduler trees*, which in turn are individually provided as input to schedulers. A scheduler, which is bound to a CPU, traverses its associated scheduler tree to give each application registered in it an opportunity to execute. At CAE, the task of building configuration files is performed by an integration specialist who relies on knowledge acquired in the past to build configurations that are valid only for a particular flight simulator.

To properly build a configuration, an integration specialist must take into account several restraints, such as applications' execution order, priority, and stringent time constraint. Applications have dependencies in the form of data exchange. In an ideal configuration, dependent applications are grouped together so that inter-processor communication is minimized, while the total load among the processors is balanced. This process is not only complex, but error-prone and time consuming. That said, the availability of an approach to automatically build configurations for flight simulators could significantly reduce the cost and time associated to this task.

In this thesis, our contribution is an approach that encompasses several steps and heuristics to automatically build configuration files for real-time aircraft simulation systems. A general overview of this contribution is presented in section 5.1. Next, we comment on future research opportunities in section 5.2.

5.1 Research Contributions

The main contribution of this study is an algorithm to automatically develop configuration files for real-time aircraft simulation systems. It is comprised of three major steps aiming at placing applications in binary trees. First, in the *placing critical applications step*, the algorithm places all critical applications together in the same binary tree, ensuring with this that they execute within the boundaries of the same processor. Next, in the *placing non-critical applications step*, non-critical applications are placed in a systematic way. An application is placed in the current scheduler tree as long as it does not exceed its capacity; otherwise, the application is placed in the next one. Finally, in the *balancing step*, a generalized ratio-cut objective function [Yeh92] is used to minimize the dependencies among resulting scheduler trees, while maximizing the total load added to each of them.

To the best of our knowledge, there is no technique designed to address our domain constraints. This is, distributing *N* applications into *K* different partitions, such that inter-partition communication is minimized, the load is balanced, and each partition is denoted as a binary tree, with finite capacity, and added semantic that allows meeting compulsory applications' priority and expected execution order.

To evaluate our approach we have conducted a case study at CAE. We have worked with CAE engineers to define two typical configuration scenarios: a 2-scheduler flight simulator, and a 3-scheduler one. We have automatically built these configurations using our algorithm. Likewise, we have asked an integration specialist to build them using the best of his knowledge. We have run the flight simulation software several times using

two simulation scenarios: one with the aircraft on ground, and another with the aircraft in air. To compare the results of *human-based* configurations against *algorithm-based* configurations, we used critical and non-critical overruns, two metrics used at CAE to evaluate the performance of the simulation. With regard to the 2-scheduler configuration, the results show that the algorithm-based configuration clearly outperformed the human-based one. The former produced more balanced partitions, and for each of them, the application's load was evenly distributed among all scheduler execution paths. With regard to the 3-scheduler configuration, the algorithm-based configuration produced better or at least similar results when compared to the human-based one in the particular case of the critical partition, this is, the one holding the critical applications. The reasons are likely the same as those for the case of the 2-scheduler configuration. For the remaining 2 partitions, the human-based configuration outperformed the algorithm-based one. Lastly, results are encouraging as in most of the cases the algorithm-based configurations outperformed or produced similar results when compared to the human-based ones.

5.2 Future Research Opportunities

To improve the results, we could include the multi-phase nature of our domain to the analysis. Flight simulation software usually executes and performs differently depending on the state or phase in which the simulation is. For instance, the engine application requires less processing time when the aircraft is on ground than when the aircraft is in air. Using this information may result in configurations that adapt to the changes in the simulation's state. It would be also possible to use this knowledge to take better decisions when building configurations that do not necessarily change over time.

Additionally, we could build on our current approach to allow for an automatic detection of the ideal number of partitions based on the inputs. This is desirable since it would produce configurations with adequate number of partitions, which in turn could help in reducing costs and deployment time.

Finally, we could also upgrade our approach to relax the penalty when dependent applications are grouped into different binary trees that are mapped to schedulers that run within the boundaries of the same PC, but bound to different CPU's. In such situations, the synchronization process required to exchange data among dependent applications running in different schedulers takes considerably less time as it does not require going over the network.

References

[Karypis98] Karypis, G. and Kumar, V., (1998), "Multilevel algorithms for Multi-Constraint Graph Partitioning," *In proc. of the 1998 ACM/IEEE Conference on Supercomputing*, Orlando, FL, USA.

[Karypis95] Karypis, G. and Kumar, V., (1995), "METIS: Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0," *Technical Report, Department of Computer Science, University of Minnesota*. Minnesota, USA.

[Schloegel02] Schloegel, K., Karypis, G. and Kumar, V., (2002), "Parallel static and dynamic multi-constraint graph partitioning," *Concurrency and Computation: Practice and Experience, 2012, 14(3),* pp. 219-240.

[Hagen92] Hagen, L. and Kahng, A.B, (1992), "New spectral methods for ratio cut partitioning and clustering," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 11(9),* pp. 1074-1085.

[FAA06] Federal Aviation Administration, (2006), "Part-60 Flight Simulation Training Device Initial and Continuing Qualification and Use," *Federal Aviation Administration, National Simulator*

Program, Federal Register, 71(209), October 30, 2006. Washington, USA.

[Kernighan70] Kernighan, B.W. and Lin, S., (1970), "An Efficient Heuristic Procedure for Partitioning Graphs", *In the Bell System Technical Journal, 49 (1970),* pp. 291-307.

[Jain10] Jain, Anil K., (2010), "Data Clustering: 50 Years Beyond K-means", *In Pattern Recognition Letters 31 (award winning papers from the 19th International Conference on Pattern Recognition, Tampa, FL, USA), 31(8), 1 June 2010,* pp. 651-666.

[Karger93] Karger, D.R., (1993), "Global min-cuts in RNC, and other ramifications of a simple min-out algorithm," *In proc. of the fourth annual ACM-SIAM Symposium on Discrete Algorithms,* Philadelphia, PA, USA, pp. 21-30.

[Luxburg07] Luxburg, Ulrike von, (2007), "A tutorial on spectral clustering," *Statistics and Computing, December 2007, 17(4),* pp. 395-416.

[Wei89] Wei, Y.-C. and Cheng, C.-K., (1989), "Towards efficient hierarchical designs by ratio cut partitioning," *In proc. of the 1989 IEEE International Conference on Computer-Aided Design,* Santa Clara, CA, USA, pp. 298-301.

[Yeh92] Yeh, C.-W., Cheng, C.-K. and Lin, T.-T.Y., (1992), "A probabilistic multicommodity-flow solution to circuit clustering

problems," *In proc. of the 1992 IEEE/ACM International Conference on Computer-Aided Design,* Santa Clara, CA, USA, pp. 428-431.

[Chan92] Chan, P.K., Schlag, M. and Zien, J., (1992), "Spectral K-Way Ratio-Cut Partitioning Part I: Preliminary Results," *Technical Report, Computer Engineering Board of Studies, University of California,* Santa Cruz, CA, USA.

[Shi00] Shi, J. and Malik, J., (2000), "Normalized Cuts and Image Segmentation," *In IEEE Transactions on Pattern Analysis and Machine Intelligence, 22(8),* pp. 888-905.

[Abramowitz72] Abramowitz, M. and Stegun, I. A., (1972.), "Stirling Numbers of the Second Kind," §24.1.4 *In Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables, 9th printing. New York: Dover*, pp. 824-825.

[Cormen09] Cormen, T.H., Leiserson, C.E., Rivest, R.L. and Stein, C., (2009), "Breadth-first search" §22.2 *In Introduction to Algorithms, third edition, The MIT Press,* pp. 594-602.

[Maini94] Maini, H., Mehrotra, K., Mohan, C. and Ranka, S., (1994), "Genetic Algorithms for graph partitioning and incremental graph partitioning," *In proc. of the 1994 ACM/IEEE Conference on Supercomputing,* Washington, DC, USA, pp. 449-457.