UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

JAVA DEVELOPMENT PLATFORM FOR REAL-TIME APPLICATIONS IN MULTI-CORE ARCHITECTURES

José Serafim Gouveia Anjos

Mestrado em Engenharia Informática

Arquitectura, Sistemas e Redes de Computadores

2009

UNIVERSIDADE DE LISBOA

Faculdade de Ciências
Departamento de Informática

JAVA DEVELOPMENT PLATFORM FOR REAL-TIME APPLICATIONS IN MULTI-CORE ARCHITECTURES

José Serafim Gouveia Anjos

DISSERTAÇÃO

Trabalho orientado pelo Prof. Dr José Manuel de Sousa de Matos Rufino
e co-orientado por Tobias Schoofs

Mestrado em Engenharia Informática
Arquitectura, Sistemas e Redes de Computadores
2009

Agradecimentos

Ao professor Rufino pelo seu empenho e dedicação.
Ao Tobias Schoofs e ao resto da familia IMA na Skysoft.

Aos meus pais, sem vocês isto não seria possível.
À minha familia por todo o apoio.
À minha irmã pelas brincadeiras.
À minha namorada pela paciência.
A todos os meus amigos.

"A todos os meus, por tudo o que vivi com vocês"

"Computers are useless. They can only give you answers."

Pablo Picasso

(113735, sdl, bod, mw, cró, sissi)

Dedicatória.

Ao meu Avô, o primeiro dos meus.

ABSTRACT

The increasing complexity that modern real-time systems are achieving has motivated many software developers to shift from the more traditional real-time used languages like C and ADA to real-time Java. Java in its earliest form seamed like irrelevant for the real-time community until the Real-Time Specification for Java changed that. The RTSJ standard aimed at creating an extension of the "Java Language specification" and "The Java Virtual machine" that allowed the creation of real-time applications using Java.

For some time now multi-core architectures have been the answer to the shortcoming showed by single-core processors. The idea to physically have parallel processing seams attractive, but the migration from current sequential processing models to parallel processing ones is not trivial.

ARINC 653 (Avionics Application Standard Software Interface) is a software specification for space and time partitioning. It defines an API for software of avionics, following the architecture of Integrated Modular Avionics. It is part of ARINC 600-Series Standards for Digital Aircraft & Flight Simulators.

This document presents a view on how it is possible to make the transition from standard real-time sequential processing applications that use more traditional languages to real-time Java applications in a multi-core architecture, by using a practical example. The point will be made by taking an existing real-time C coded application that is a avionic communication software that uses the ARINC 653 standard. The idea is to make a Java version of the application and a parallel processing version and prove that not only it is possible to do so, but it brings several benefits.

To support this document it will be used the research that the JEOPARD project is making along with the tools that the project is developing.

KEYWORDS:

Java ; Real-time ; Multi – Core ; Parallel Processing ; Development platform ; ARINC 653

RESUMO

É com pouca surpresa que verificamos que o nível de complexidade dos sistemas tem vindo a aumentar expressivamente. Actualmente faz parte do quotidiano utilizar serviços, online ou não, que atingem nível de complexidades muito superiores quando comparados aos sistemas de à 20, 30 anos atrás. Sistemas/Aplicações de tempo real são quase como uma subgrupo dentro dos sistemas actuais. Sistemas de tempo real podem ser encontrados nos mais diversos locais, actualmente quase todos os sistemas embebidos conteem componentes de tempo real. Alguns exemplos de sistemas que podem utilizar tempo real são, carros, radares, braços robóticos, linhas de montagem, satélites, UAV(*Unmanned Aerial Vehicle*) e até aviões.

Geralmente para implementar sistemas de tempo real eram utilizadas linguagens consideradas mais tradicionais, tais como C ou ADA. No entanto o nível de complexidade tem aumentado tanto que os programadores têm procurado alternativas a essas linguagens. Uma das alternativas encontradas veio de uma linguagem já existente e extremamente popular, Java. Java é uma linguagem de programação desenvolvida pela Sun Microsystems. Conta actualmente com cerca de 6,5 milhões de programadores e pode ser encontrada em virtualmente todos os segmentos de indústria. Tudo isto faz do Java uma linguagem extremamente atractiva.

Devido ao seu propósito e desenho quando Java foi introduzido em 1995 parecia irrelevante no campo da computação de tempo real. Apesar de boas características introduzidas pelo Java no desenvolvimento de software simplesmente não era compatível com sistemas de tempo real. No entanto devido a popularidade que o Java ganhou ao longo dos anos, em 1998 um grupo de programados juntou-se para criar o Real-Time Specification for Java (RTSJ).

O Real-Time Specification for Java é uma série de interfaces e especificações que permite que se desenvolva sistemas de tempo real na linguagem de programação Java. O objectivo é fornecer informações complementares que permitam, entre outras coisas, estender o modelo de memória Java e melhorar a semântica do escalonador de threads. O RTSJ pretende criar uma extensão da especificação da linguagem Java e da sua máquina virtual.

As arquitecturas multi-core são uma das possíveis soluções para o processamento paralelo. Multi-core tem sido a tecnologia de ponta há já alguns anos. A grande motivação por detrás do multi-core tem sido as limitações que os processadores de um único core apresentam. Um processador multi-core é um processador que combina mais que um core independente. Cada core pode implementar optimizações tais como multi-threading. Um sistema que apresente n cores vê a sua performance optimizada quando se vê perante n threads a funcionar concorrentemente. Em termos gerais cada core num processador multi-core assemelha-se a uma implementação de um processador em single-core.

A ideia de ter paralelismo a um nível físico é uma ideia extremamente atractiva. Se conseguirmos dividir uma tarefa grande em várias pequenas e executar essas pequenas tarefas em simultâneo conseguimos obter resultados extremamente satisfatórios. No entanto por muito vantajoso que seja a utilização de modelos que utilizem paralelismo, a migração dos modelos sequenciais para modelos paralelos não é de todo trivial. Nem todas as tarefas são facilmente divididas em várias pequenas tarefas, e nas tarefas paralelas existem por vezes problemas com execuções concorrentes.

Para obter com sucesso uma aplicação de tempo real é necessário suporte para o sistema, para isto é necessário um RTOS (Real-Time Operating System). Um RTOS é um sistema operativo que se destina a sistemas/aplicações de tempo-real. Tal como todos os sistemas de tempo-real, um RTOS oferece uma camada entre o software e os recursos de hardware.

Espera-se que uma aplicação de tempo-real a correr num RTOS tenha um comportamento que encaixe em padrões temporal pré definido. Isto significa que a quantidade de tempo que cada serviço do sistema é conhecido e restrito.

Apenas o RTOS não garante que um sistema seja capaz de cumprir os prazos que a aplicação requer. É da responsabilidade de quem está a desenvolver o sistema/aplicação fazer com que esse sistema/aplicação seja capaz de cumprir os prazos pretendidos.

Neste sentido o objectivo principal de um RTOS é de fornecer um ambiente de execução que consiga suportar serviços para sistemas/aplicações de tempo real.

O ARINC 653 (Avionics Application Standard Software Interface) é uma especificação para o espaço e partição temporal. Define uma API de aviação que segue a

arquitectura IMA (Integrated Modular Avionics). O sistema IMA suporta uma ou mais aplicações de aviação em apenas um módulo, um módulo é uma máquina que contem aplicações ARINC 653, e permite a execução independente dessas aplicações. Isto pode ser atingido caso o sistema forneça separação funcional, normalmente chamada de partição robusta, de aplicações de aviação. A especificação faz parte da serie ARINC 600-Series Standards for Digital Aircraft & Flight Simulator.

Este documento apresenta uma visão de como é possível fazer a transição de aplicações de tempo real sequenciais que utilizem linguagens mais tradicionais para aplicações de tempo real em arquitecturas multi-core utilizando Java. Para este efeito uma aplicação já existente vai ser utilizada. A aplicação é denominada como "Airline Operational Centre" ou AOC. O "Airline Operational Centre" foi desenvolvido pela Skysoft e é utilizado como uma solução de comunicação para aviões. O seu principal objectivo passa pela distribuição de relatórios entre as estações em terra (AGP - AOC Ground Platform) e o sistema no avião. O sistema dentro do avião é constituído por dois intervenientes, o próprio avião que vai fornecendo informação relevante como temperaturas, consumo de combustível etc., e o piloto. A aplicação AOC é uma aplicação ARINC653 e como tal segue a filosofia por detrás do conceito IMA. A aplicação AOC foi totalmente implementada utilizando a linguagem de programação C, e segue modelos de processamento sequenciais. A ideia por detrás da validação da aplicação passa por fazer não só uma versão mais simplificada do AOC original, mas também uma versão em Java e outra versão que utilize processamento paralelo. Com estes novos modelos esperamos provar que é possível implementar um AOC em Java que utilize processamento paralelo e que isso trás vários benefícios.

O objectivo estratégico do projecto JEOPARD é fornecer ferramentas para o desenvolvimento de sistemas previsíveis que façam uso de arquitecturas multi-core. Estas ferramentas vão melhorar a produtividade do software e a sua reciclagem através da extensão da tecnologia de processador já estabelecida em sistemas desktop para as necessidades específicas de sistemas embebidos em arquitecturas multi-core.

O projecto vai activamente contribuir com as normas requeridas para o desenvolvimento de software portável neste domínio, tal como o Real-Time Specification for Java.

Também, o projecto JEOPARD vai desenvolver uma plataforma de desenvolvimento para aplicações de tempo real em ambientes multi-core. A interface vai ser baseada em tecnológias existentes, que incluem o the Real-Time Specification for Java (JSR 1 and JSR 282) e o Safety-Critical Java (JSR 302). Estas tecnologias actualmente fornecem uma forte base para o desenvolvimento de sistemas de tempo real que sejam complexos e altamente fiáveis, mas ainda não fornecem suporte para sistemas multi-core. Ainda mais desafiante, é o facto de que algumas destas tecnologias não referem o facto de se poder utilizar mais que um core de cada vez, o que torna impossível desenvolver aplicações à escala com o número de processadores disponíveis em sistemas multi-core actuais e futuros.

Como suporte a este documento vai ser utilizada a pesquisa que o projecto JEOPARD está a fazer como também as ferramentas que o projecto está a desenvolver.

PALAVRAS CHAVE:

Java ; Tempo real ; Multi-Core ; Processamento paralelo ; Plataforma de desenvolvimento ; ARINC 653 ;

Content

List of Figures

List of Tables

Chapter 1 Introduction

This chapter provides a first view over the problem to be discussed in this dissertation, the motivation that drives it forward and its objectives. The structure that offers support for the application development in Real-Time Java language in multi-core architectures and working environment is also presented.

1.1 Motivation and Objectives

We are faced with a software crisis. Gordon E. Moore in a 1965 paper [1] stated that:

"The complexity for minimum component costs has increased at a rate of roughly a factor of two per year ... Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years. That means by 1975, the number of components per integrated circuit for minimum cost will be 65,000. I believe that such a large circuit can be built on a single wafer."

And yet on April 13th 2005, Gordon Moore stated in an interview that the law cannot be sustained indefinitely [2]:

"In terms of size (of transistor) you can see that we're approaching the size of atoms which is a fundamental barrier, but it'll be two or three generations before we get that far—but that's as far out as we've ever been able to see. We have another 10 to 20 years before we reach a fundamental limit. By then they'll be able to make bigger chips and have transistor budgets in the billions."

As stated by Moore in 2005 the fundamental limit, that is the atom, is fast approaching. Add to this problem the growing demand of processing power with restrained power consumption that embedded computers demand. Solutions must be

found to prevent us from reaching a evolutionary cul-de-sac in technological achievements in the processing area.

Such a solution is through the use of parallel processing models on software development processes in a multi-core architecture [3]. But for that solution to work we must begin to think outside the sequential models of software development and embrace parallel designs that will allow us to come closer to use all the potential that is in multi-core architectures. Such a passage from sequential to parallel is not trivial. Most of the software developers are not prepared to begin developing parallel applications and there is a lack of tools to directly support multi-core embedded software development.

Java has for some time enjoyed a high popularity level, due to its properties like being purely object-oriented and highly portable. Its familiar syntax with C++ and user friendly memory management provided by the garbage collector only increases even further the levels of productivity achieved and thus its popularity.

Some of the properties that make Java so popular also make it improper for real-time development, like the presence of an automatic memory management form, known as garbage collection and the poorly specified thread scheduling. The Real-Time specification for Java (RTSJ) [4] and later the Safety Critical Java Technology (SCJT) [5] were the answers that came forth to allow the development of Real-Time applications. Real-time specification for Java introduced some challenges in its implementation in a real-time java virtual machine [6]. On top of that the RTSJ does not account for the use of parallel computing [7].

The JEOPARD project and the work developed within the scope of this dissertation aims at filling the gap between real-time applications written in Java and multi-core architectures.

The main focus of the problem to be addressed consists in the passage of a sequential, C coded, ARINC653 application to a parallel processing, Java coded, ARINC653 application in a multi-code architecture. The application being an ARINC653 application follows the philosophy of the IMA (Integrated Modular Avionics) architecture.

The thesis will study different facets regarding the passage of the application, like the best way to make the passage from sequential to parallel, how to make the best use of the multi-core architecture, and the actual gains (by measurement) that a java-parallel-multi-core architecture has in comparison with the original C-non parallel - non multi-core application.

The work developed under the scope of this thesis benefits from several past and present projects. The application that will be used for the validation of the development platform is provided by the MA-AFAS (The More Autonomous Aircraft in the Future Air Traffic Management System). The application is an avionic software used for communication between airplanes (AOC- Airline Operational Center) and the ground station (AGP – AOC Ground Platform). The focus for this work will be on the airplane (AOC) part of the software.

Since the AOC is an ARINC-653 application it needs to run in a native ARINC-653 operating system. The AMOBA project and its simulators provides an inexpensive and realistic way of functionally testing ARINC-653 applications.

1.2 Company Description

The work developed under the scope of this thesis was performed under a proposal of Skysoft Portugal SA

1.2.1 Skysoft Portugal SA

Located at Lisbon's new business centre, in Parque das Nações, and home to a team of highly qualified professionals, Skysoft Portugal is a technology and systems provider in the areas of Aeronautics, Security, Space and Transports.

Since its foundation Skysoft has taken pride in cooperating and working with some of the major European agencies and industry players in those fields, as well as in playing

an important role in both national and international consortia for the development of break-through projects.

Nearly a decade of experience has solidified Skysoft success and proved its capacity for creating innovative solutions for different types of missions.

Skysoft Portugal acknowledges that keeping up to date with the progress of cutting-edge technology is the key to the future, and relies on continuous research activities and in a talented, skilled and motivated team to keep its major commitment of offering innovation - through the development and use of value-added technologies - to its clients and partners.

The work discussed in this dissertation was mainly developed under the scope of the JEOPARD project, being carried out in the Aeronautics Security, Space and Transports Division of the Skysoft Portugal.

1.3 JEOPARD Project Overview

JEOPARD, the Java EnvirOnment for ParAllel Real-time Development is an European project funded by the 7th Framework programme of the European commission. The project is run by the Open Group (UK) and it's partners are Aicas (Germany), EADS (Germany), SysGo (France), RadioLabs (Italy), Forschungszentrum Informatik (Germany), University of York (UK), University of Cluj-Napoca (Romenia), University of Vienna (Austria) and Skysoft (Portugal). The JEOPARD project will develop a platform independent software development interface for complex multi-core systems, focusing on SMP (Symmetric Multiprocessing).

"The strategic objective of the JEOPARD project is to provide the tools for platform-independent development of predictable systems that make use of SMP multi-core platforms. These tools will enhance software productivity and reusability by extending processor technology already established on desktop systems for the specific needs of multi-core embedded systems. The project will actively contribute to standards

*required for the development of portable software in this domain, such as the Real-Time Specification for Java (RTSJ)."(*www.JEOPARD.org)

By using the JEOPARD platform and its tools in the re-programming of a real-time, sequential processing and complex application into a parallel processing one in a multi-core architecture, Skysoft will help evaluate the platform as a valid support tool for the implementation of parallel real-time java applications.

1.4 Related Projects

Several projects serve as base to the work developed. The most relevant is the MA-AFAS project that provides the application that will be used as test subject. The AMOBA project will provide a platform to run the test application in a ARINC-653 environment. The AIR and DIANA projects serve as reference since the objective of both the projects is to build ARINC-653 development platforms.

MA-AFAS

The More Autonomous Aircraft in the Future Air Traffic Management System (MA-AFAS) project, funded by the European Commission through the 5th Framework Programme, was part of a larger European undertaking of reducing delays by improving means for aircraft movement control in the European airspace. It aimed to transform european research results into practical operational Air Traffic Management (ATM) procedures. One of the applications, developed in this context, was the Airline Operational Centre (AOC) solution.

AMOBA

The ARINC 653 simulator for modular space-based applications is an ESA co-funded project, developed by Skysoft that aims to provide system engineers with an execution environment capable of executing and testing ARINC 653 applications [8, 9,10]. The AMOBA simulator will provide an effective environment for the development of avionics and aerospace applications and verify their behavior without the need to access a real ARINC 653 real-time operating system (RTOS- Real Time Operating System) or their final hardware infrastructure [11].

AIR

AIR (ARINC 653 in RTEMS (Real-Time Executive for Multiprocessor Systems)) was a project funded by ESA and developed by Skysoft Portugal SA and Faculdade de Ciências da Universidade de Lisboa (FCUL). The idea behind AIR was to provide RTEMS with the functionalities and interface specified by the ARINC 653 specification. This gave RTEMS a development environment that enabled it to run avionic applications in full conformity with the ARINC 653 standard [12].

AIR – II (ARINC 653 in Space RTOS) is a continuation of the AIR project, it is funded by ESA and developed by Skysoft Portugal SA, FCUL and Thales Alenia Space. The idea is to prove the feasibility of an ARINC 653 compliant RTOS to be used in space. It did this by proposing a RTOS independent architecture that allows different executives to be used as Partition Operating System.

DIANA

Diana (Distributed Equipment Independent Environment for Advanced Avionic Applications) is a R&D project that aims in creating an enhanced avionics platform

called AIDA (Architecture for Independent Distributed Avionics). The project is conducted by a consortium that is coordinated by Skysoft Portugal SA.

The Diana project proposes the use of a middleware layer in the AIDA platform called "Neutral Execution Platform" that enables to run not only standard ARINC653 applications but also Object Oriented (OO) applications. The AIDA JVM (Java Virtual Machine) that the Diana has uses the Safety Critical Java Technology [13].

1.5 Document organization

This document is organized as follows:

1. Chapter 2 – **State of the art –** offers an insight view of the current technological state of the technologies that will be used on the work developed under the scope of this dissertation.
2. Chapter 3 – **Conception Requirements –** will give a more detailed view of the problems and solutions found while trying to achieve the main objective.
3. Chapter 4 – **Application to multi-core and Java as proof of concept –** presents the different solutions for the Real-Time Airline Operational Centre (AOC) application that is being studied. This along with the results that were obtained with the models and discussion of theirs meaning.
4. Chapter 5 – **Conclusions and Future Work** – presents the conclusions drawn from this study along with the possible future work that can be developed.

1.6 Publications

The work developed under the scope of this dissertation has contributed to the following publication in an international conference.

An Integrated Modular Avionics Development Environment: T. Schoofs, S. Santos, C. Tatibana, J. Anjos, J. Rufino and J, Windsor, In Proceedings of the DASIA 2009 – DAta Systems In Aerospace Conference, EUROSPACE, Istanbul, Turkey. May 2009.

The publication refers to the development environment called IMADE (Integrated Modular Avionics Development Environment) that has the AMOBA simulator as a key component of the development environment. The work developed under the scope of this thesis helped to validate the AMOBA simulator as a valid ARINC 653 environment simulator.

Chapter 2 State of the art

This chapter is dedicated to the overview of the state of the art of elevant technologies. Since the target application of our study is an ARINC 653 based application that is to be ported to a Java multi-core environment. The technologies in question are all used to improve the application, either by performance (parallel processing, multi-core) to use a more friendly language (Java) and to keep the real-time nature of the original application (real-time).

2.1 - Real-Time

Imagine jumping out of an airplane with a parachute that guarantees that it will open every single time. In the air you press the button and nothing happens. Three minutes after you hit the ground the parachute opens. It did open, but it did you no good… *"A real-time computer system is a computer system in which the correctness of the system behavior depends not only on the logical results of the computations, but also on the physical instant at which these results are produced."[Hermann Kopetz, Real-Time Systems]*

The work being developed under the scope of this thesis is highly connected and dependent of the concept of Real-Time. The main objective focused on the demonstration that a path leading from standard Real-Time applications to Java Real-Time applications on multi-core architectures is not only possible but very beneficial. The use of the Java Virtual Machine provided by the JEOPARD project and its validation can only be achieved by using a Real-Time application like the AOC application . All of these reasons make the concept of Real-Time the core component of the work being developed for this dissertation.

It is usual to have a real-time system to have different time requirements. The same system may have tasks that must be completed in a specific time frame, or tasks that do not require any constraints concerning time frames. Real-time systems may be categorized as such:

Within the Real-Time world there are several particular cases. In the realm of avionics one of such particular case is the IMA (Integrated Modular Avionics).

2.1.1 - Integrated Modular Avionics

IMA systems support one or more avionics applications hosted on one module (a machine that contains ARINC 653 partitions) and allows independent execution of these applications. This can be achieved if the system provides a functional separation, usually called robust partitioning, of the avionics applications, for fault containment, such that a failure in one separated function cannot cause a failure in another partitioned function; in consequence the partitioning approach eases validation, verification and certification. The unit of partitioning is called a partition. A partition is basically the same as a program in a single application environment: it comprises data, code and its own context and configuration attributes (see figure 1).

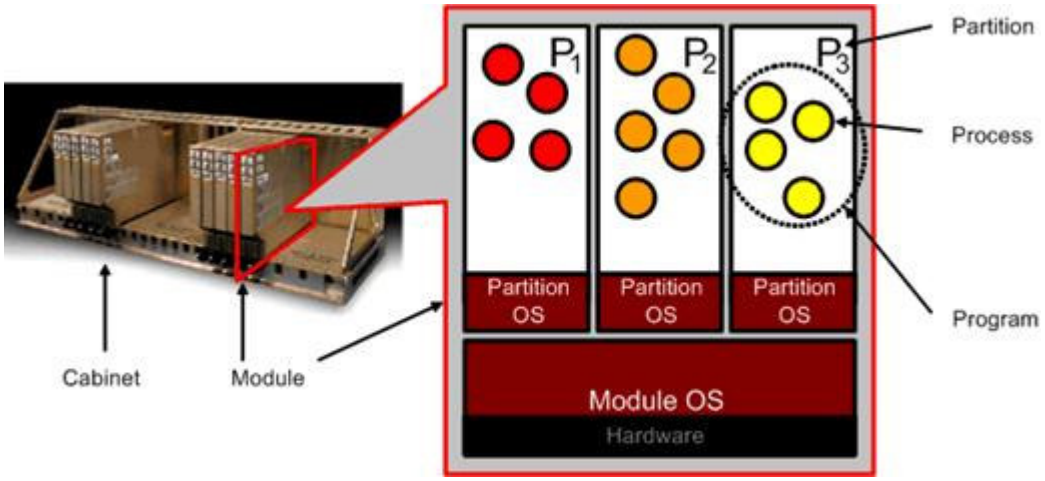

Figure 1- Partitioning in IMA

Partitioning separates applications in two dimensions: space and time. Spatial separation means that the memory of a partition is protected. No application can access memory out of the scope of its own partition. Temporal separation means that only one application at a time has access to system resources, including the processor; therefore only one application is executing at one point in time – there is no competition for system resources between partitioned applications.

The ARINC 653 specification defines a static configuration where each partition is assigned a set of execution windows. The program in the partition associated with the current execution window gains access to the processor. When the execution window terminates, the program is pre-empted; when the next execution window starts, the partition associated to this execution window will gain access to the processor as illustrated in Figure 2.

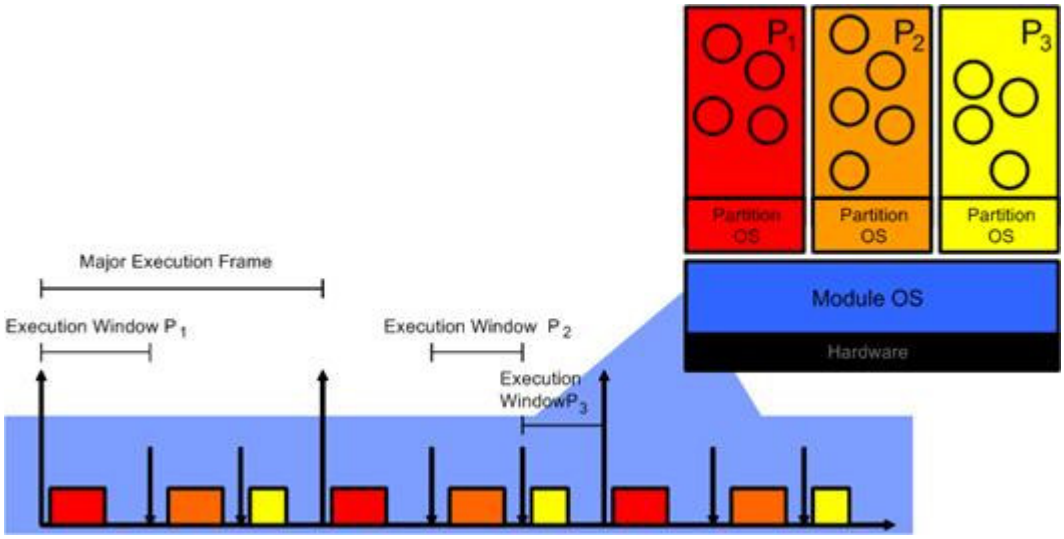

Figure 2 - Time Partitioning

Processes within the scope of a partition are scheduled by a priority-based pre-emptive scheduler with FIFO (First-In First-Out) order for processes with the same priority. This second level scheduler is invoked whenever an execution window assigned to its partition starts and the partition gains access to the processor. It is pre-empted by the first level partition scheduler when the execution window terminates.

ARINC 653 defines an API (Application Programming Interface) providing applications with a set of services. These services are partition-internal inter-process

communication means, like events, message buffers, black boards and semaphores; time services; an interface to the health monitor as illustrated in Figure 3.

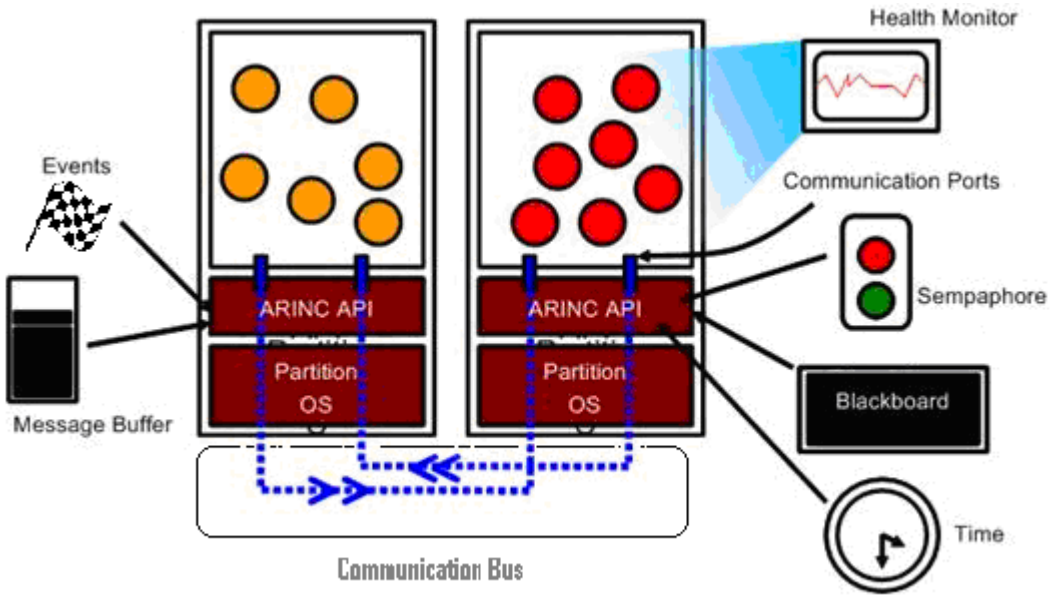

Figure 3- ARINC 653 API

The communication among partitions occurs through ports defined at partition level. A port is either incoming or outgoing; two behaviors are defined: a port may be a queue holding zero or more messages or a so called sample port containing the current instance of a periodically updated message [14].

2.2 – Modern Language: JAVA

Java is a programming language developed by Sun Microsystems. With 6.5 million software developers Java is an attractive language. Java can currently be found in virtually every industry segment.

Known for its versatility, efficiency, platform portability and security, it currently powers 4.5 billion devices [15].

It was originally designed to have the following goals:

- Simple, Object Oriented, and Familiar – Java aimed at providing a simpler yet familiar language, by maintaining a syntax similar to C and by making it a object oriented language.
- Robust and Secure – By raising the level of abstraction of low level issues, like memory addresses and pointers, Java is more robust and secure than other languages like C.
- Architecture Neutral and Portable – One of the main advantages of Java is its portability, the same Java code works on every platform that has a Java virtual machine.
- High Performance – Despite all the advantages that using Java brings, it must maintain a high performance.
- Interpreted, Threaded, and Dynamic – By using byte-codes the Java interpreter can execute on any machine. Multithreading is supported at the language level. The language and run-time system are dynamic when it comes to its linking stages.

Due to its purpose and design when Java was introduced in 1995 it seamed irrelevant in the field of real-time computing. Despite the good characteristics introduced by Java for software development it is not compatible with real-time embedded systems, due to the presence of garbage collection and the poorly specified thread scheduling.

Due to the popularity that Java gained through the years, in the winter of 1998 JSR – 000001 was proposed [16].

Real-Time Specification for Java (RTSJ) is a set of interfaces and behavioral specifications that allow for Real-Time programming in the Java programming language. The aim is to provide additions, like extending the Java memory model and improving the semantic thread scheduling.

When dealing with concurrent programming we are faced with two units of execution; threads and processes.

A process is an instance of the execution of a computer program that usually contains a self-contained execution environment and complete run-time resources; each process contains its own memory space.

A thread also called lightweight process differs in some aspects from processes, threads exist only as a subset of a process, and while each process has its own address space threads share their address space. Processes can have several threads that share state, memory as well as other resources.

When concurrent programming is performed using Java it is greatly done using threads. With a Java Virtual Machine RTSJ compliant, Real-Time Java can be achieved by the use of two classes introduced by the RTSJ interface, the RealtimeThread and the NoHeapRealtimeThread. Both classes extend the Thread class. Therefore when referring to real-time Java programming one can only mention threads as the concurrent programming unit. This document follows this statement.

"I expect the "The Real-Time Specification for Java" to become the first real-time programming language to be both commercially and technologically successful."[4]

Real-Time Group

2.3 Parallel processing Infrastructures

With the development of the RTSJ the development of real-time java applications became a possibility, but the RTSJ does not contemplate the use of parallel processing models. The limited processor capacity available and the potential gains that parallel processing offers, a evolution of current standard applications to parallel models must be made. That's where the JEOPARD and this thesis contributions really stands out.

"I know how to make 4 horses pull a cart - I don't know how to make 1024 chickens do it."

Enrico Clementi

Traditionally, computer software has been made by using sequential processing models. When resolving a problem, an algorithm is planned, implemented and used as a serial stream of instructions. These instructions are executed in a single central processing unit. Only one instruction is executed at a time and once that instruction was finished the next would have its turn and so on.

Parallel computing is the simultaneous use of multiple compute resources to solve a computational problem. It is to be run using multiple processors based on the principle that large problems can be divided into smaller independent problems. By dividing the problem it is possible to carry out the calculations of each smaller problem simultaneously by concurrent processing (parallel processing).

Using multiple processors operating together on a single problem is not a new idea; in fact it is a very old idea. Gill writes in 1958:

"... There is therefore nothing new in the idea of parallel programming, but its application to computers. The author cannot believe that there will be any insuperable difficulty in extending it to computers. It is not to be expected that the necessary programming techniques will be worked out overnight. Much experimenting remains to be done. After all, the techniques that are commonly used in programming today were only won at the cost of considerable toil several years ago. In fact the advent of parallel programming may do something to revive the pioneering spirit in programming which seems at the present to be degenerating into a rather dull and routine occupation..."[17]

Sequential and parallel systems can roughly be categorized as stated in Flynn's taxonomy illustrated in figure 4. [43]

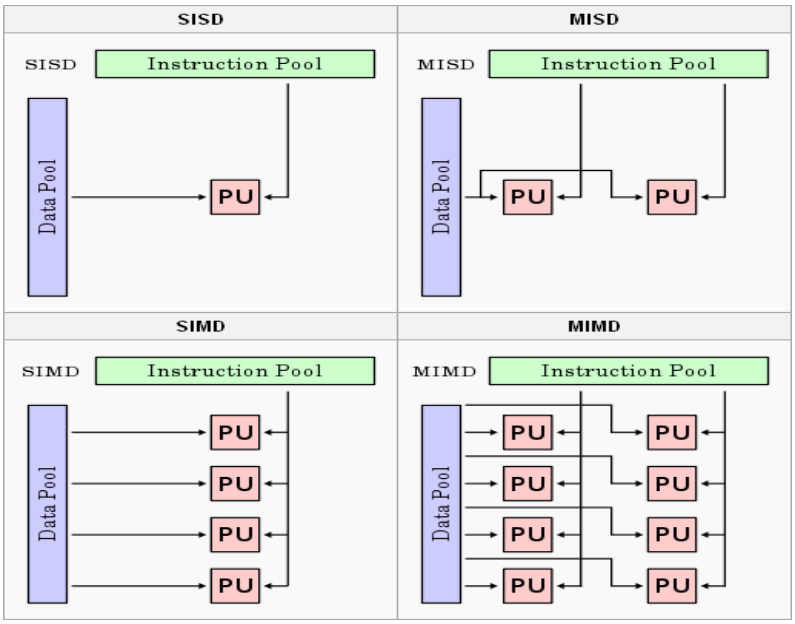

Figure 4 - Flynn's taxonomy

PU – Processing Unit

Single Instruction, Single Data stream (SISD) - Follows the traditional sequential processing models.

Single Instruction, Multiple Data streams (SIMD) - By using a single instruction that is exploited by multiple data streams it performs operations that are "embarrassingly parallel".

Multiple Instructions, Single Data stream (MISD) - Multiple instructions operate on a single data stream. Uncommon architecture which is generally used for fault tolerance.

Multiple Instructions, Multiple Data streams (MIMD) - Multiple autonomous processors simultaneously executing different instructions on different data. This architecture is usually associated with distributed systems, by using either shared memory space or distributed memory space.

For the work being developed for the dissertation, the application (AOC) that originally uses SISD architecture will be adapted to a parallel processing model. The bases for the architecture that the parallel processing version of the AOC, is based in the architectures presented in the Figure 4. Although the final solution presented in this dissertation for the application AOC may not statically be equal to any of the models, it will be heavily based on them.

Why parallel processing

"Imagine the task of digging 12 holes. If one man takes 1 hour, 2 should take 30 minutes, 3 should take 20 minutes and so on."

The main reasons to use parallel computing began with the need to save time and money. Computational requirements are ever increasing, both in the area of scientific and business computing. In theory by adding more resources to a task will shorten the required time for its completion, and with potential cost savings. High-performance parallel computers clusters can be made out of inexpensive hardware, like Beowulf clusters. [18]

The complexity of some problems makes them impractical or even impossible to solve with a single computer. The time it takes to complete a task or limited memory that is at the disposal of a single computer make extremely difficult to resolve problems that require PetaFLOPS and PetaBytes of computing resources.

Sequential architectures are reaching a physical limitation. The level of miniaturization possible for a number of transistors to be placed on a chip will eventually reach a limit. [2] The speed with which a serial computer can achieve is directly dependent on the used hardware. This along with the increasing difficulty with heat and power consumptions has made progression increasingly less feasible. [19]

Current computer architectures are increasingly relying upon hardware level parallelism to improve performance:

- Multiple execution units
- Pipelined instructions
- Multi-core

2.4 - Multi-Core: a modern parallel processing architecture

As stated multi-core is one of the possible solutions for parallel processing. Multi-core architectures have been the state-of-the-art for some years now. This was motivated by the limitations that single core processors imposed on developers [19].

A multi-core processor is a processor that combines more than one independent core. Each core independently implements optimizations such as multithreading. Multithreading simply put is a way to increase utilization of a single core by leveraging thread-level as well as instruction-level parallelism. A system with n cores is effective when it is presented with n or more threads concurrently.

In general, each core in a multi-core processor resembles a single-core processor implementation. The implementation of the cache hierarchy in a dual-core or multi-core processor may be the same or different from the cache hierarchy implementation in a single-core processor.

Two threads executing on two cores in a multi-core processor can take advantage of the shared second-level cache, accessing a single-copy of cached data without generating bus traffic. The Second-level cache (L2) can be seen in multi-core architectures like the ones depicted in Figure 5 and 6.

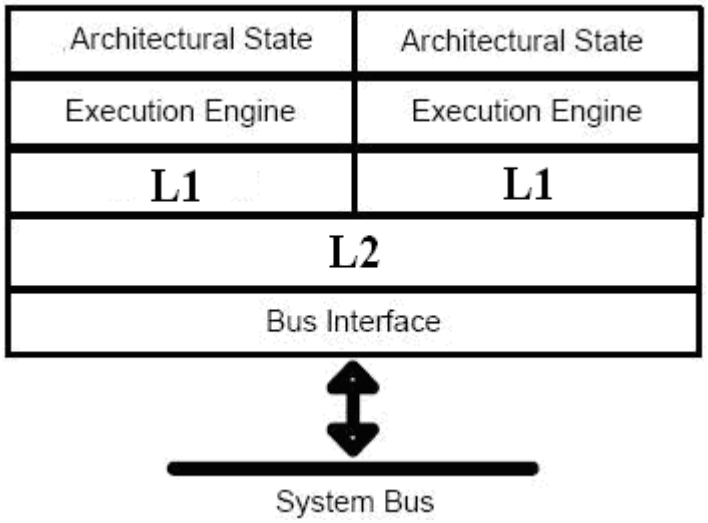

Figure 5– Dual-core processor architecture

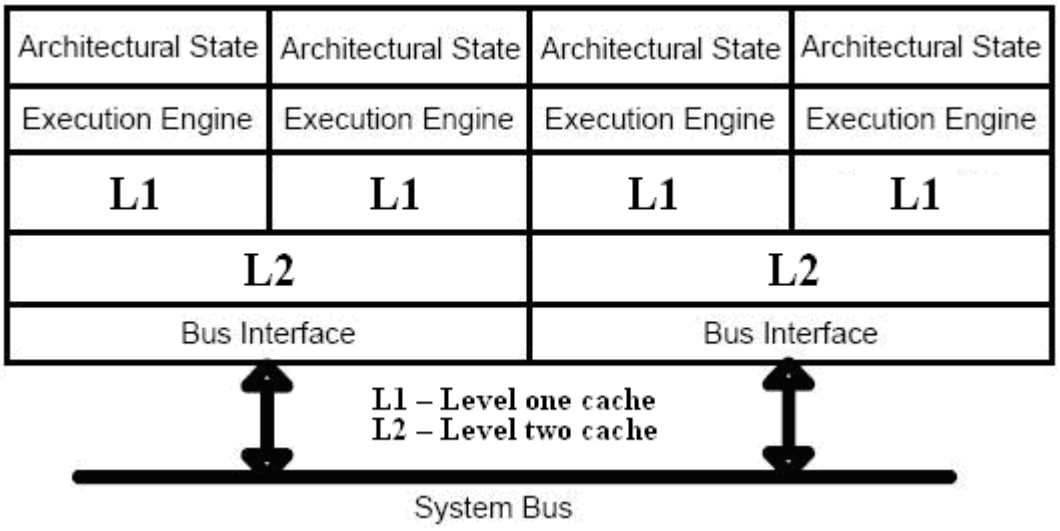

Figure 6– Quad-core processor architecture

Figure 5 and 6 represent a dual-core architecture and a quad-core architecture respectively. This quad-core architecture represents one of the earliest models were it is simply two dual-core processors put together. Although this is not an optimal solution its benefits are visible. We now have twice as many cores as in the dual-core architecture.

When an instruction needs to read data from a memory address, the processor looks for it in cache and memory. When an instruction writes data to a memory location the processor first makes sure that the cache line that constrains the memory location is

owned by the first-level data cache of the initiating core. Then the processor looks for the cache line in the cache and memory sub-systems.

The cache line is taken from the L1(level one cache) data cache of the other core only if it is modified, ignoring the cache line availability or state in the L2 (level two cache) cache [20].

The amount of performance gained by the use of a multi-core processor depends on the problem being solved and the algorithms used, as well as their implementation in software [21].

The fast development of new multi-core architectures shows us that we cannot limit ourselves to the current technologies such as the dual and quad core shown above; soon an 80 core processor will be upon us [22]. And there is no end in sight. To this effect new approaches to multi-core architectures are taking place. One of such approaches is the use of asymmetric multi-core architectures, what this means is that inside the same die we find cores with the same support for instruction sets but their performance will vary. The reason for the difference in performance may be the result of different clock speed, different hardware characteristics and different cache access times.

It is predicted that processors with tens or even hundred of cores will come into existence in the next decade [23] increasing their performance. As such the main reason for the development of heterogeneous multi-core architectures finds its motivation in the fact that it is necessary to find a way to make the increased core number processors more cost efficient. Such an approach is shown as viable [24, 25, 26].

Two additional concepts are important when dealing with multi-core architectures, simultaneous multithreading or hyperthreading (using the Intel term) and CPU (Central Processing Unit) affinity. CPU affinity simply put is a modification of the scheduling algorithm. The thread or process in the waiting queue simply contains a field that indicates the processor that it wishes to be allocated with in preference of all the others. When a processor has hyperthreading it is treated by the operating system as having two cores. This occurs as a hardware simulation, only one physical core is represent by is treated as two virtual processors. This enables the operating system to schedule two processes or threads simultaneously on the same processor. Hyperthreading is possible by

duplicating certain parts of the processor, like those that store the architectural-state, but not the main execution resources.

2.5 - Real-Time Operating System

A RTOS (real-time operating system) is an operating system intended for real-time application/systems. Like any other operating system it provides a layer between software applications and hardware resources.

A real-time application running in a RTOS is expected to have deterministic behavior. That means the amount of time each system service uses is known and bounded. The RTOS alone does not guarantee that a system is able to meet its deadlines. The developer of the system must also build a system that is compliant with the intended deadlines.

The RTOS main objective is to provide an execution environment as well as support services to real-time systems/applications.

The services provided by a real-time operating system are:

Task-Manager & Scheduler

RTOS provides the ability to divide the application into different execution units, called tasks. Tasks are equivalent to threads in the Java scope.

Real-Time is about deterministic behavior rather than speed, in this context the task-manager is responsible for insuring that the resources (like processor time) are used in the correct way to ensure the tasks achieve timeliness guarantees.

Time Management

Due to the real-time requirements of the RTOS some services must be provided to help with the time specifications of each task. Services like time-outs and task delays.

Memory Management

Like most operating systems RTOS provide mechanisms with which is possible to dynamically manipulate memory (e.g. the allocation of variables). Dynamic allocation of variable memory buffers is incompatible with real-time.

Inter-Task Communication

Most RTOS provide a variety of mechanisms for communication and synchronization between tasks. These mechanisms are especially important when dealing with a task preemptive environment. This makes possible for tasks to coordinate between each other and cooperate with other tasks.

Input/Output Management

This service is responsible for every handling read, write peripheral requests for the system. It does this by using device drivers or Interrupt Manager specific for the hardware device in concern.

2.6 – Operating Systems Support

The application to be addressed within the scope of this thesis requires a operating system with real-time capabilities. There are several real-time platforms available, choosing one is closely dependent on the needs of the system.

Some of the possible solutions will be discussed below.

IMA and ARINC653 Operating System

It defines an API for software of avionics, following the architecture of Integrated Modular Avionics. It is part of ARINC 600-Series Standards for Digital Aircraft & Flight Simulators.

For a RTOS to be a ARINC653 Operating System it basically needs the following concepts a time space partitioning kernel that supplies the required services stated in the ARINC653 standard [27, 28] and the APEX that supplies the interface.

The Standard architecture for a ARINC 653 architecture is illustrated in figure 7.

The APEX layer provides the ARINC 653 interface between the core software layer and the application software layer.

The core software layer implements partitioning that provides the services used by the API and behaviors that are specified by the ARINC 653 specification, and the system specific functions that may include hardware interfaces like system drivers.

The application software layer is composed by application partitions and system partitions.

1. Application partitions are segments of code dedicated to avionic applications. They are restricted to the ARINC 653 calls for any interface with the system.

2. System partitions are similar to application partition except they are outside the scope of the APEX layer. They are able to directly interface with the system specific function. As a consequence they are optional and specific to core module implementation.

All partitions are constrained by spatial and temporal partitioning.

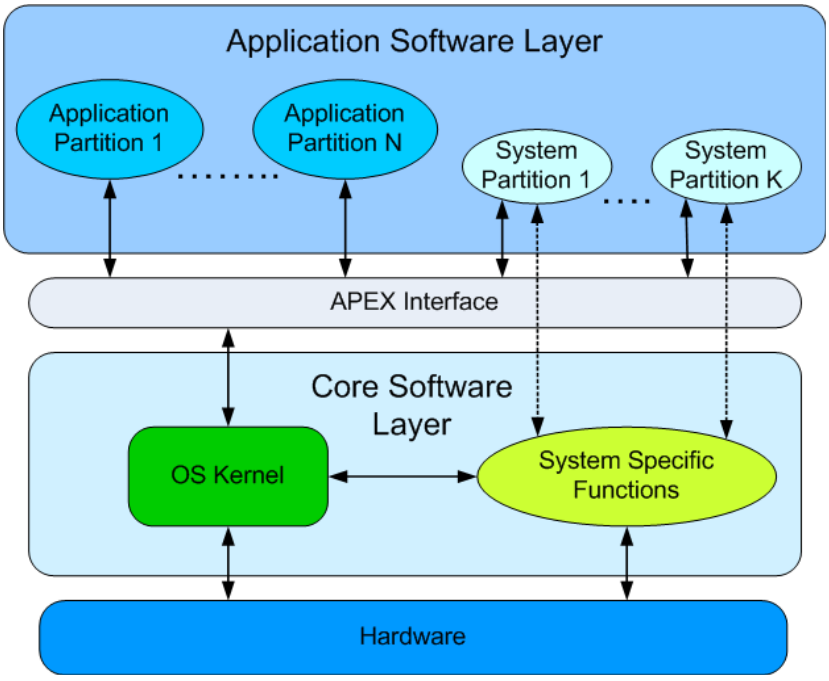

Figure 7- Core Module Component Relationships

RTEMS

RTEMS (Real-Time Executive for Multiprocessor Systems) is a RTOS developed by OAR Corporation, it is a open source and designed for embedded systems.

RTEMS is designed to support several open API standards including POSIX while providing support for applications with real-time requirements.

RTEMS provides multitasking capabilities either in homogeneous or heterogeneous architectures. It is totally event-driven possessing a preemptive scheduling and is priority-based.

PikeOS

PikeOS was developed by SysGo to be an efficient paravirtualization real-time operating system based on a separation microkernel.

The PikeOS separates by address spaces every driver, real-time application, stacks and hosted OSs with pre-defined I/O access. With paravirtualization PikeOS allows the combination of applications with different levels of security and safety, all in the same platform. It offers the possibility to host a variety of operating systems APIs and runtime environments [29, 30]. Thus allows PikeOS to run applications based on standards like POSIX and ARINC 653 as illustrated in figure 8.

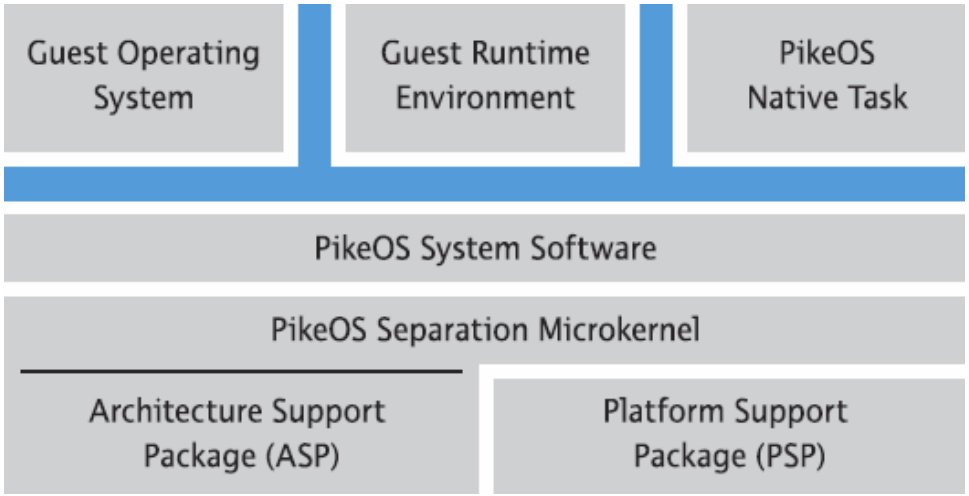

Figure 8- PikeOS architecture on a single core

Real time in the standard Linux kernel

For the work being developed under the scope of this thesis it is crucial to have a hard Real-Time environment that is able to run the AMOBA simulator and allows to run our applications with Real-Time. The Linux operating system acts as a ideal candidate for this work, it only needs to have hard Real-Time capabilities.

The previous two architectures only supplied means to run a Linux-Kernel alongside real-time tasks. For a Linux Kernel to support real-time it has to undergo some significant alterations. The main obstacle in achieving real-time performance in a Linux kernel is linked with the lack of preempted calls that user space processes make at the kernel. Thus if a lower priority task makes a system call, the higher priority task must wait until the task finishes so it can gain access to hardware resources.

Soft real-time with standard Linux kernel

With POSIX Real-Time interface it is possible to make soft Real-Time applications, but without guarantees.

Through a simple kernel configuration it is possible to achieve soft real-time performance in the 2.6 Linux kernel. This configuration enables the kernel to preempt higher priority tasks to run, even if any system call is running. The configuration is not without its downsides, the kernel experiences a break in its performance and a lower throughput due to the added overhead done by the configuration.

Hard real-time with standard Linux kernel

"Controlling a laser with Linux is crazy, but everyone in this room is crazy in his own way. So if you want to use Linux to control an industrial welding laser, I have no problem with your using PREEMPT_RT."

Linus Torvalds
Developer of the
Linux kernel

The PREEMPT_RT patch was originally developed by Thomas Gleixner and Ingo Molnar. The patch is directly applied in a standard Linux kernel and it aims in providing the Linux Kernel with hard real-time capabilities. It reduces the amount of code in the kernel that is non-preemptible and the code that must be altered to add preemptibility.

Chapter 3 – Conception Requirements

In this chapter a more detailed view of the thesis work and its objectives will be discussed as well as a more detailed view on the JEOPARD project. A section will be devoted to the difficulties and solutions that comes with using real-time with Java, namely RTSJ and SCJT, and later with adding multi-core architectures.

3.1 Analysis Requirements

With the growing popularity of the Java programming language and the standardization of multi-core architectures as a parallel processing solution, it was just a matter of time for the real-time community to join the evolution.

The development of technologies such as RTSJ and SCJT made possible the creation of real-time compliant Java virtual machine with supporting API's such as Sun's real-time Java Virtual Machine and IBM's Websphere. But none of these solutions allows the applications to use the potential that comes with parallel processing in multi-core architectures; the main reason is that RTSJ does not address the issue of using parallel computing.

The main objective of the work developed under the scope of this thesis is to show that the evolution from standard Real-Time applications to Java Real-Time applications with parallel processing in a multi-core architecture is possible and brings several advantages.

The objective will be validated by a demonstration of a practical use case. An existing application was chosen to serve as a demonstrator. The application is the Airline Operational Centre (AOC). The AOC is the avionics systems interface to a communication facility that serves to exchange a set of messages, called reports, between an airplane and a ground station. The original design of AOC is an IMA application

compliant to ARINC 653 and as such follows the IMA philosophy and the rules in the ARINC 53 specification [27]. The application is processed sequentially and totally made in C language.

In order for the validation to be made it is necessary to compare the original application with its Java and parallel versions. The validation is the analysis of the time that the AOC takes to produce the reports and to process a certain number of reports. In this scenario it is expected that Java will achieve a worse performance versus the original application due to the properties of the Java language, and that the parallel processing version using a multi-core architecture will achieve an increased performance in the processing of several reports simultaneously despite the added overhead that comes with the parallel model.

In order for the comparison to be made a Java version of the AOC and a parallel processing version must be made. The implementation of the Java version of the AOC requires a passage from the entire C code application to a Java coded application while maintaining the original structure. For the parallel version of the AOC a study of the application has to be made in order to find the best suited parallel processing model for our particular case. The resulting parallel processing on a multi-core architecture application will not be completely ARINC 653 compliant. According with the standard any partition must be treated like a multi-tasking application, each partition consists of one or more concurrently executing processes, sharing access to processor resources based upon the requirements of the application. Also the processes of a partition, consequently the partition itself, cannot be divided over multiple processors. Since the AOC is a partition it cannot have its processes divided between several cores. It is this contradiction that dictates that the resulting parallel processing using multi-core AOC application does not completely follow the ARINC 653 standard.

The project will have the support of the JEOPARD project, more specifically the tools that are being developed by the project.

3.2 JEOPARD Project

The strategic objective of the JEOPARD project is to provide the tools for platform-independent development of predictable systems that make use of multi-core platforms. These tools will enhance software productivity and reusability by extending processor technology already established on desktop systems for the specific needs of multi-core embedded systems. The project will actively contribute to standards required for the development of portable software in this domain, such as the Real-Time Specification for Java.

In addition, the JEOPARD project will develop a platform-independent software development interface for real-time multi-core systems. The interface will be based on existing technologies, including the Real-Time Specification for Java (JSR 1 and JSR 282) and Safety-Critical Java (JSR 302). These technologies currently provide a strong foundation for the development of complex and highly reliable real-time systems, but they do not yet provide support for multi-core systems. Even more challenging, some of the technologies can not address more than one processor at a time, making it impossible to develop applications that scale with the number of processors available on current and future multi-core systems.

Skysoft has the task (as a JEOPARD partner) to make the assessment of the platform through an application program or re-programmed in Java. This also includes a study the benefits of multi-core architecture and the Java language in this context. These objectives are addressed under the present thesis.

3.3 Real-Time & Java

The increasing demand for more complex real-time systems has triggered a search for a more “friendly” programming language in oppose the traditional real-time development languages like Assembly, Ada and C. Java as complexity goes seems like a

prime candidate. But Java is not a language suited for the development of real-time applications due to such problems like a non preemptible garbage collection mechanism and a poorly specified thread scheduling. The Real-time Specification for Java (RTSJ) stepped up to provide the means to give Java what it needs to become a viable choice as a real-time development language.

3.3.1 Real-Time Specification for Java

By 1998 three groups led their own investigations to provide a Java platform with support for real-time programming. The groups were:

- IBM, Greg Bollella led a group of companies with an interest in Java and real-time.
- NIST (National Institute for Standards and Technology) led by Kelvin Nielsen and Lisa Carnahan.
- Sun, with a project based in their Java platform.

In that same year the three groups merged and the result was the *"Requirements for Real-Time Extensions for the Java Platform"*, that provided with the requirements the group had developed.

In late 1998 Sun was asked to resolve a JSR (Java Specification Request) the *"Real-Time Specification for Java"* using as base the work developed by the group so far. Sun deemed the request as JSR-000001 with Greg Bollella as its leader.

The group aimed at creating an extension of the *"Java Language specification"* and *"The Java Virtual machine"* together with an application programming interface (API) for real-time threads. The specification adds two new classes of threads to the API, these threads exist to ensure the real-time requirements, they are:

- Real-Time Threads (RTT) are those who are placed in a memory area that allows thread logic access to instance variables and methods normally.

- No-Heap Real-Time Threads (NHRTT) are an extension to Real-Time Threads. The main difference is that NHRTT do not have access to heap memory, without access to the heap these threads are not disturbed by the garbage collection.

The RTSJ features support for deadlines together with CPU time budgets, scheduling properties that suit real-time application requirements and tools to prevent tasks of receive delays provided by the garbage collection. Due to these features the RTSJ supports soft and hard real-time applications [4].

The specification is guided by seven rules, the rules are as such:

1. RTSJ is not restricted to any particular Java environment.
2. RTSJ does not prevent former Java applications to run on implementations of the RTSJ.
3. Recognition for “Write Once, Run Everywhere” although considering the possibility of sacrificing binary portability in exchange for predictability.
4. RTSJ addresses the current practice concerning real-time systems but it allows future implementation for any future features.
5. The first and main priority of the RTSJ is the predictability of its executions.
6. The non-introduction of any new keywords.
7. The RTSJ allows the variation in implementing decisions, thus giving flexibility to create implementations to better suit the need of the requirements.

"My biggest fear is that people will confuse real time with real fast."

Greg Bollella
Distinguished Engineer
Sun Microsystems

3.3.2 Safety Critical Java Technology

The JEOPARD project is based not only in the RTSJ but also in the SCJT technologies.

The SCJT (Safety Critical Java Technology) is a work in progress. Made to allow Java to be used in safety and mission critical applications due to the exceedingly rigorous validation and certification (of DO-178B Level A certification) processes that are needed. The SCJT is derived from the RTSJ and uses the real-time extensions and determinism behavior that the RTSJ introduced. The specification proposes a much simpler version of the memory model presented by the RTSJ. Five ground rules support this decision, they are:

1. All and any approach due to predictable resistance to the acceptance of a new programming model of general asynchronous transfer of controls, even if proved predictable and safe, should be a conservative one.
2. All real-time extensions written in the SCJT must be at the very least a subset of the RTSJ. Therefore all applications done using the SCJT should run under the RTSJ.
3. If functionalities are missing in the current version of the RTSJ request should be made to the TIC (Technical Interpretation Committee).
4. New classes may be added to the API only if they ca be made using RTSJ.
5. Annotations can be used to allow off-line activities.

The SCJT will support a reduced standard of the Java API, this is made to support the applications through the process of validation and certification. The specification will not use all of the classes presented in the RTSJ and some of the ones that are used will have restrictions. SCJT will also propose some new classes in its specification [5].

SCJT presents some differences in application development that shifts away from the RTSJ.

SCJT presents the notion of “mission” Mission has three different phases:

- Initialization, that is single threaded. In this phase all schedulable objects are created.
- Mission, starts after initialization ends, all objects that were created in the initialization phase are now ready for execution.
- Termination marks the end of the mission phase, where all the threads stop and the application ends.

It is possible to have a fourth phase called “Recovery” where the initialization and mission phases are run again. This is to facilitate the possibility of execution of series of missions as well to correct any failure that might have occurred.

3.3.3 Real-Time & Java – Challenges and Standard Solutions

RTSJ introduces several new APIs in such areas as:

- Threads
- Memory Management
- Synchronization
- Time and Clocks
- Asynchrony

Together with these new APIs, add the need for more predictability in the execution environment and all of this amounts to having several challenges in implementing a RTSJ-compliant Java virtual machine.

The RTSJ has requirements that need support from the OS, like priority inheritance. So the first challenge is to find a suitable OS where the JVM can run. Some solutions are available, solutions like using the Linux kernel plus PREEMPT_RT patch.

When comparing a standard JVM with a RTSJ-Compliant JVM some “incompatibilities” stand out. Standard JVM have worker threads to run some supporting

tasks like the garbage collector (GC) or the Just-In-Time-Compiler (JIT). The problem here is the priorities with which the worker threads are running, if they are not correctly set they can interfere with the real-time threads that are running. By setting the priority of the worker thread responsible for the JIT to be below the lowest priority possible of the RTT or NHRTT we ensure that it will not interfere with the real-time properties. In the case of the GC worker thread it is not so simple since the GC needs to have permissions above the RTT but below the NHRTT.

One of the specifications on the RTSJ that is not found in standard JVM is that every reference object that is being read or stored must be checked. This is made to make sure that the No-Heap Real-Time threads (NHRTT) do not access heap allocated objects and that the Scope reference rules are enforced. This is a strait foreword implementation problem.

It is important to manage the structures and resources that must be used by both Real-Time Threads (RTT) and NHRTT to ensure that the NHRTT do not get disturbed by the garbage collection.

Another issue connected with implementing RTSJ in a JVM has to do with performance, all the modifications that have to be made will add to the overhead that may affect the performance of the JVM [33].

3.4 Real-Time & Java & Multi-Core

For the real-time community the shift from conventional to more modern languages like Java presented several challenges, these challenges were addressed and solutions were proposed in the form of the RTSJ document. Multi-core has become a standard in modern processors as an answer to the problems encountered while trying to develop faster single core processors[19]. Like any significant development it is progressive as such the RTSJ does not address the issue of using multi-core architectures in the development of Java coded real-time applications. The fact of the matter is that introducing a new architecture like multi-core to real-time systems, where predictability

is the key word, is not without its challenges. The study and solutions found by the JEOPARD project shed some light in this part of the real-time development, the Java coded real-time with multi-core architectures.

3.4.1 Real-Time & Java & Multi-Core – Difficulties and Solutions

The task of providing a RTSJ compliant Java virtual machine with the capability to take advantage of multi-core architectures presents several challenges. Several of these challenges will be discussed below.

To introduce parallel computation together with the current version of the RTSJ some problems appear. We have some low level issues regarding performance.

When introducing a new processor architecture like multi-core it is inevitable not to be faced with some performance issues. Issues like memory access, single thread running time and event handling.

Regarding memory management if we have two cooperating threads running in distinct cores and if these cores are connected by on-chip caches (L2 or L3 caches) then a core does not have to go to the system bus on account of a cache miss, it can simply go to a on-chip cache for something that the other core already did. If the two threads run on different processor chips then the overhead gained depends on the access type.

RTSJ requires support for CPU affinity, the possible gains come from that over time data from the process/threads may remain in the local core cache and thus the performance is improved. Modern processors can exhibit two architectural features that can influence the overall performance is the existence of simultaneous multithreading (SMT) or hyperthreading (Intel terminology) and asymmetric processing.

Second there are high-level issues such has [7]:

Co-scheduling – Co-scheduling is a mechanism proposed for concurrent systems that schedules related processes to run on different processors at the same time.

When dealing with multiprocessor systems we are presented with two types of scheduling. Temporal were the objective is to keep all the processors busy, improving the

overall throughput, and spatial scheduling were the objective is to improve a certain application by executing simultaneously its threads/processes.

For multi-core co-scheduling may be a desirable feature, having for instance four threads that need to be co-scheduled in order to improve the systems behavior. The downside of co-scheduling on Real-Time systems is that it may conflict with the priority-based scheduling. If four threads are running in a four processor system and the four threads need to be co-scheduled, but a fifth and with higher priority is runnable. In this scenario the rest of the three processors will wither stall or execute threads with lower priority.

The solution is given by the existence of a global scheduler, but global solutions have poor performance when compared with local solutions. So when designing a scheduler that supports co-scheduling in multi-core architectures the best course of action is to make one that has a balance between the need to share global knowledge and the overall performance of the solution.

Asymmetric processing – Asymmetric processing despite the advantages it brings for processor manufactures it brings several disadvantages for programmers. The levels of unpredictability that asymmetric processing architectures have are not acceptable in the real-time community. For this effect an asymmetry-aware scheduler is needed.

Chapter conclusion

With the work developed in the RTSJ it was possible to make Real-Time Java Virtual Machines, and the classes to develop Real-Time Java applications. This together with the study that is being made to provide these virtual machines with the ability to use multi-core architectures it is natural that a course from standard Real-Time applications to Real-Time Java in multi-core architecture is plotted. In the next chapter we take a real case of a real-time application and prove that that course is viable and presents advantages.

Chapter 4 – Application to Multi-Core and Java Proof of Concept

In this chapter a description of the AOC application will be presented as well as its architecture and its units for the measurements in order to perform its validation. It will also be presented two other AOC models; the Java AOC and the Parallel AOC. The results of the measurements of the three models will be compared and subsequently discussed in this chapter.

In this chapter there are two concepts that may look similar but are in fact very different. The AOC System is the entire application it refers to all of the partitions and its communication system. The AOC partition is the core of the application it is the partition that makes the computation of the "events" that will be explained in the next section. When referring only to AOC or AOC application one must assume that it's the AOC System that is referred.

4.1 – Airline Operational Centre

In this section it will be described in detail the AOC system its architecture and the times measured in the system and its significance. The AOC system will spawn two more models, a Java model and a parallel model. It's the comparison between the original application against the Java and parallel models that will provide the values for the results.

4.1.1 - Airline Operational Centre Application

AOC is a avionic software developed by Skysoft that is used as a communication solution. Its main objective is the distribution of reports between ground stations (AGP) and the on-board system (pilot and airplane). It is a ARINC 653 application and as such follows the IMA concept. The AOC is part of a larger system, see figure 9.

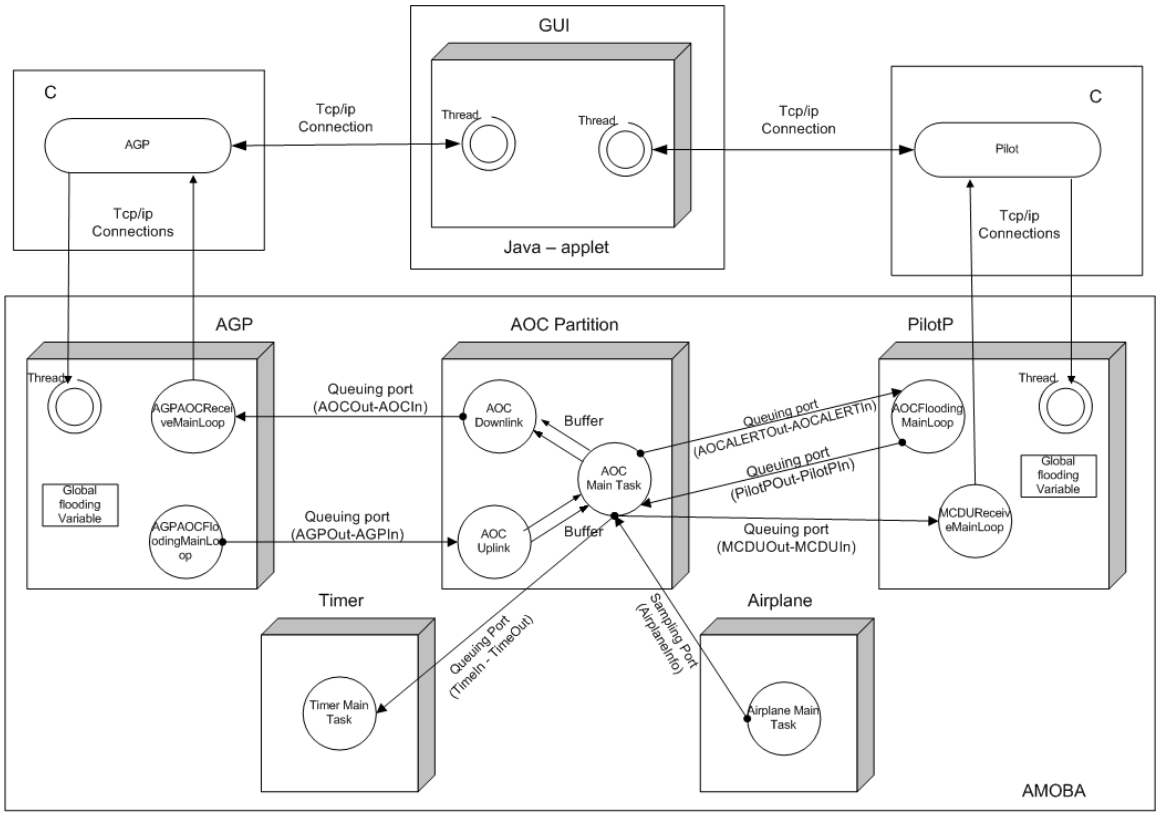

Figure 9– AOC System architecture

The GUI (Java) is the controller of the entire system. It manages the messages received/sent by the pilot and AGP. The Pilot and AGP (programmed in C language) are simple applications made to be the "middleman" between the GUI and the AOC system.

The AOC is comprised of 5 partitions:

1. The AGP is the partition that is responsible for receiving messages either from the GUI or the AOC partition and forwarding them.
2. The Pilot executes a similar function as the AGP partition but for the pilot.
3. The Airplane is a partition that mimics the existence of an airplane sending general information (Oil temperature, fuel flow etc.).
4. The Timer partition exists only to receive the processing times of the AOC partition.

The AOC partition is the most important partition in the AOC system, it is responsible for receiving, storing, retrieving and sending messages. Every time it receives a message from the AGP the process stores the messages, sends an alert to the Pilot and waits for a request from the Pilot to send it. It also receives messages from the Pilot partition creates the appropriate report and relays it to the AGP. It is the center of all the communication of the application.

The AOC partition contains a periodic service that is used to send a report to the AGP every second. This is part of the hard real-time specification within the application.

The AOC system is both data and process intensive and as such it is a good candidate to be ported to Java because it uses complex data structures. It is also a good candidate to execute a migration to a parallel processing style since it consists of tasks that can be executed in parallel. The AOC partition for each iteration begins with a list of messages it must process. As a sequential application it must process a message and only then it may begin processing another.

The Figure 10 represents the processing logic of the AOC partition.

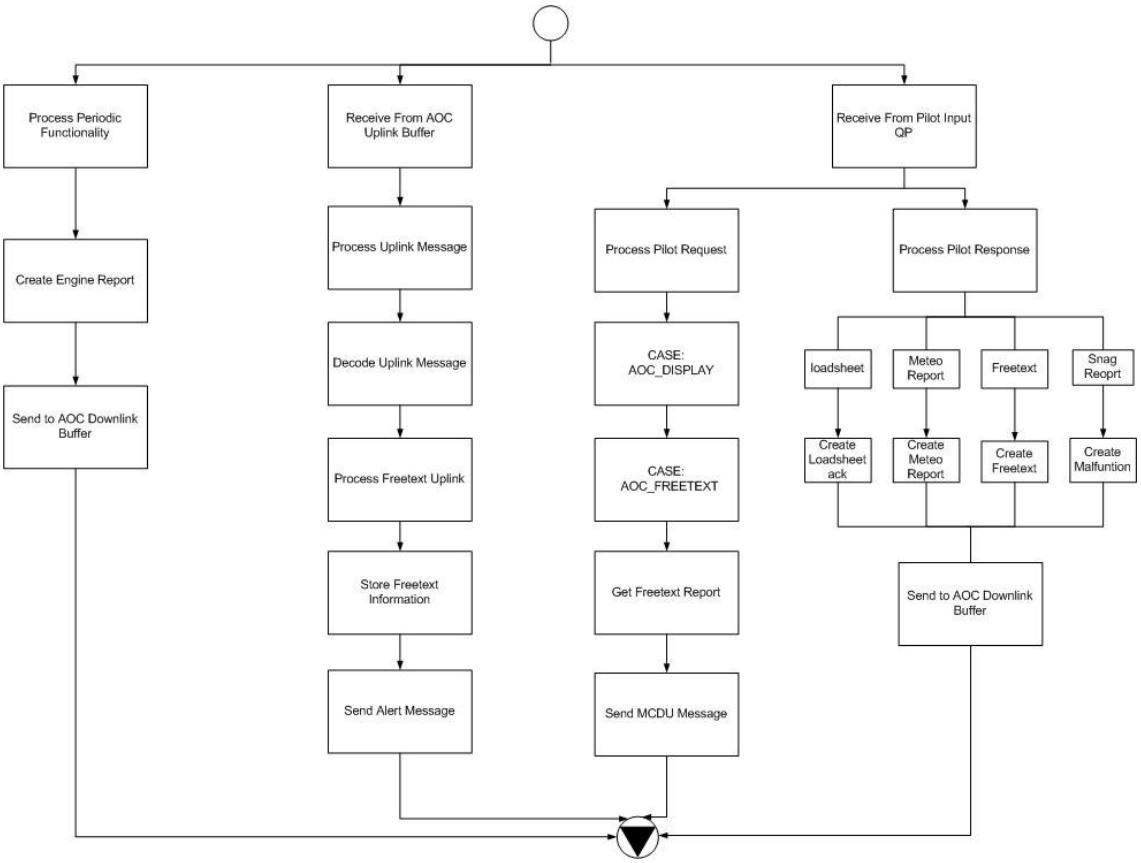

Figure 10– AOC partition message processing

The AOC system being a communication solution depends heavily on inter-partition communication. Every action that is performed by the AOC partition (except for the periodic functionality) is an event triggered by the reception of a message (either by the Pilot or the AGP) that ends with the sending of a message.

In the system there are eight relevant events that will serve for the validation of the application. They are summarized in Table 1.

Name	When it occurs?	What is measured?
Create Free Text Downlink Message	The pilot sends a message to the AOC, the AOC creates the appropriate report and sends it to the AOC.	The time it takes the AOC to produce a Free Text Downlink Message.
Create Malfunction Message	The pilot sends a message to the AOC, the AOC creates the appropriate report and sends it to the AOC.	The time it takes the AOC to produce a Malfunction Message.
Create Load sheet Acknowledgment.	The pilot sends a message to the AOC, the AOC creates the appropriate report and sends it to the AOC.	The time it takes the AOC to produce a Load sheet acknowledgment.
Create Meteo Request	The pilot sends a message to the AOC, the AOC creates the appropriate report and sends it to the AOC.	The time it takes the AOC to produce a Load sheet acknowledgment.
Process Periodic Functionality	Every second the AOC produces a engine report and sends it to the AGP.	The time it takes to produce the engine report.
Process Pilot Request	After the pilot receives a message that the AOC has a stored message to the pilot. The pilot requests the stored message.	The time it takes for the AOC to retrieve the message to produce the report and to send it to the pilot.
Process Pilot Response	It's the event that triggers all of the create reports. The pilot sends a message to the AOC to be sent to the AGP.	The time it takes for the AOC to produce the report and send it to the AGP.
Process Uplink Message	The AGP sends a message to the AOC to be stored until the Pilot requests it.	The time it takes the AOC to store the message and send an alert to the Pilot.

Table 1– List of AOC Partition Events

The main expected difference in gain between the three different models is going to be measured by the number of events it can process in a given time period. With this objective in mind it is possible to activate the flooding capability in the AGP and Pilot partitions. The flooding causes the two partitions to "flood" the AOC partitions with requests and responses and causes the AOC partition to overflow, this makes the AOC partition to work at its full capacity. In the Java model is also interesting to compare the time it takes to process the different eight events with the C model to see how much slower Java is when compared with C.

For all models the partitions will respect a scheduling as the ARINC 653 specification requires. The major time frame for all the partitions is of 1 second, and the partitions run for:

AOC partition:	0.3 seconds
Pilot partition:	0.2 seconds
AGP partition:	0.2 seconds
Airplane partition:	0.2 seconds
Timer Partition:	0.1 seconds
TOTAL	1 second

Table 2- AOC Scheduling times

The times used for the application are loosely based in actual avionic communication application times. The order which the partitions run is the same as the order presented in table 2.

For the Java and parallel models some things must be maintained:

- All the partitions in the original AOC system (except the AOC partition) must be equal for every model, and must also work in the same way for all the models. This includes the scheduling times, see table 2.
- The AOC partition that processes reports must maintain its processing logic as depicted in figure 10. In the case of the parallel model all the partitions must maintain this processing logic.
- Every partition (partitions in the case of the parallel model) must run for 0.3 seconds and do nothing for 0.7 seconds (as is defined in the ARINC 653 configuration for the original AOC application), this is done to allow all the models to be on equal footing with each other.
- Every AOC models must measure the same events as defined in the Table 1.
- All the five partitions defined in the original AOC system must run in a single core.

4.1.2 – Airline Operational Centre to validation application

For demonstration purposes the AOC application will have to be simplified as shown in figure 9. The communication is made by using UDP and TCP communication.

For demonstration purposes the number of messages will be significantly reduced since there is no need for such a large number of messages to be processed given that the application will be used to provide an example of the feasibility of the proposed objective.

The AOC application in the scope of this thesis can be separated into Real-time component and non Real-time. The application submits a report every second which is a hard Real-time specification within the application, on the other hand there is the existence of aperiodic reports that are in the non Real-Time category within the application. This means that these aperiodic reports have no time requirements. The Real-time component is the most important one and the one most relevant for the tests. As such some of the aspects that should be measured are:

1. Average processing time per report
2. Worst case processing time per report
3. Minimum processing time per report
4. Maximal number of reports processed within the given time period.

The average processing time per report is interesting, in particular, with respect to aperiodic reports. For real-time applications, the worst case processing time is much more important than the average time. However, in AOC partition aperiodic reports will be processed with lower priority than periodic. In consequence a big amount of periodic reports will decrease the amount of possible aperiodic messages: the communication via

aperiodic reports will slow down. Therefore, increasing the average performance of real-time processing will allow more non/soft-real-time communication.

The time measuring process is done as such:

```
Time a = getTimeStamp();
EVENT
Time b = getTimeStamp();
```

The resulting subtraction (b-a) gives you the time that the EVENT took.

The most noticeable changes regarding the original AOC application are:

- The Pilot and AGP will be simple application that stand for the pilot and ground station being able to send and receive messages.
- The Airplane is a simple application used to feed data into the AOC partition simulating the conditions changing in the airplane.
- The Pilot and AGP are partitions that act like middle man in the communication of the pilot and ground station with the AOC partition.
- The AOC partition is the core of the System, much like what was described in the section 4.1.1.

4.1.3 – Airline Operational Centre system Architecture

The AOC system is a multi-partition application that depends heavily on inter-partition communication as stated in section 4.1.1 and needs to be in an ARINC 653 compliant environment.

The AMOBA layer will enable the AOC to run in a ARINC 653 environment even if the operating system does not provide support for the standard [34].

The chosen operating system is the Linux + RT Patch; with this OS we guarantee hard real-time capabilities in a familiar OS capable of running the AMOBA simulator.

The Hardware used is a standard Intel Quad core processor, although only one core will be used to test this version of the AOC system.

The figure 11 better depicts the original AOC system architecture.

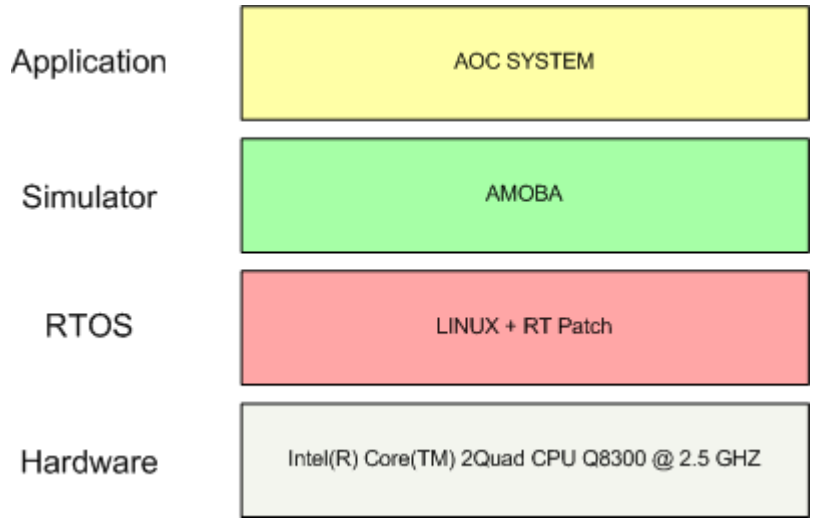

Figure 11– AOC Architecture

4.2 – Java version for Airline Operational Centre Application

This section describes the second solution of the AOC application, the Java model presents a Java AOC partition that runs in a real-time Java virtual machine and uses the RTSJ specification to achieve hard real-time.

4.2.1 – Java Airline Operational Centre Application

For this model a duplicate of the AOC partition was made using the Java programming language. The Java AOC partition is in (almost) every way identical to its C counterpart. The Java AOC partition has its main differences in the extensive use of classes and objects (something that does not occur in the C AOC partition) and in the lack

of an APEX interface. The Java partition still maintains the processing logic, see figure 10 and the same events as before, see Table 2.

In this model the AOC system will maintain the scheme and architecture that the original AOC system has and adds the Java AOC partition to the system outside the scope of the AMOBA layer, as depicted in figure 12.

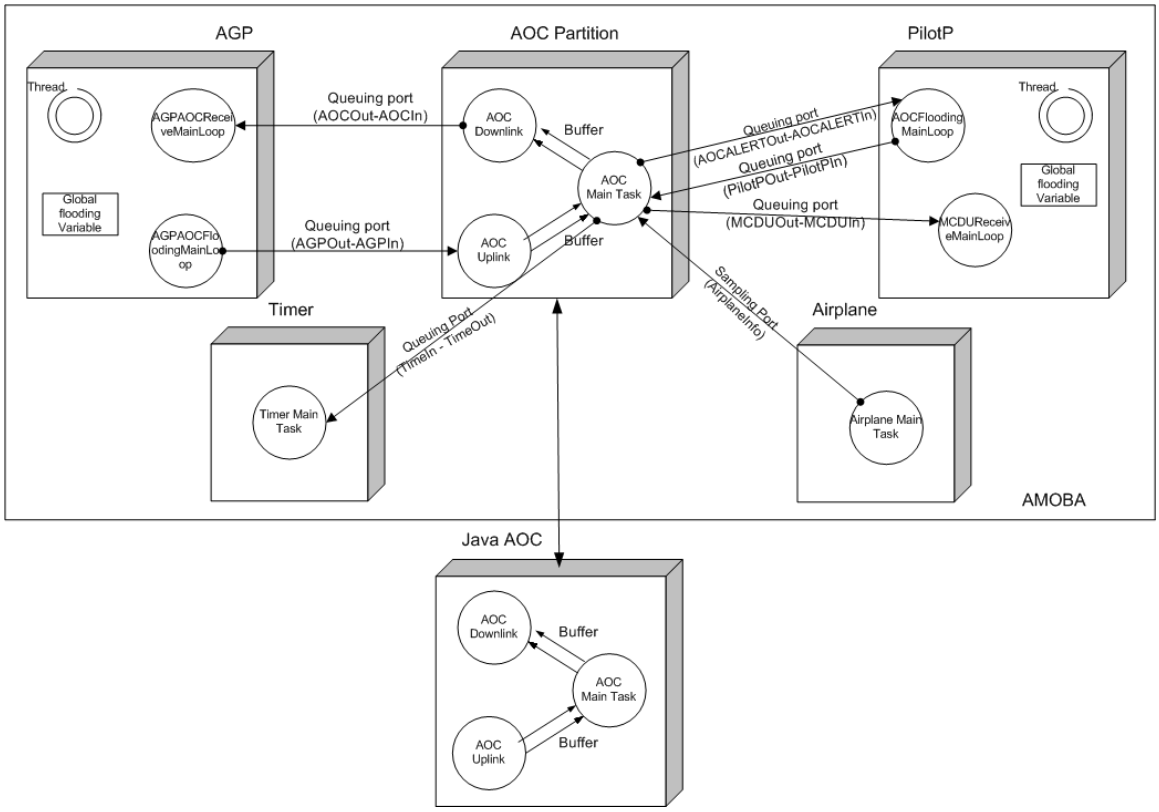

Figure 12– Java AOC System

The logic behind this model is that all the AOC Partition inside the AMOBA is that serves only as a relay partition, directing all messages received to the Java AOC and correctly sending to the correct partitions all the messages received from the Java AOC.

With this system all the properties of the original AOC system are maintained, such as the rate that the requests and responses reports that the PilotP and AGP make to the AOC. As well as the ARINC 653 scheduling model defined in the original AOC system. In this model the AOC partition that falls into the AMOBA scope simply receives the reports from the AGP and PilotP and relays them to the Java AOC partition. The Java

AOC partition receives the reports, processes them and sends them back to the AOC partition in the AMOBA layer; it then relays them to the correct partition.

Like the original AOC partition the Java AOC partition only runs for 0.3 seconds, this ensures that the processing time the new partition has is equal to that the original partition had.

This new model adds overhead to the entire system but also enables the AOC model to maintain its original structure.

The time measurements are all done in the Java AOC Partition in the same way as in the original AOC partition by use of the Timer partition.

4.2.2 – Java Airline Operational Centre Architecture

The almost identical structure of the Java model with the original model allows it to maintain an almost identical architecture with one vital difference, the inclusion of a real-time Java virtual machine, see figure 13.

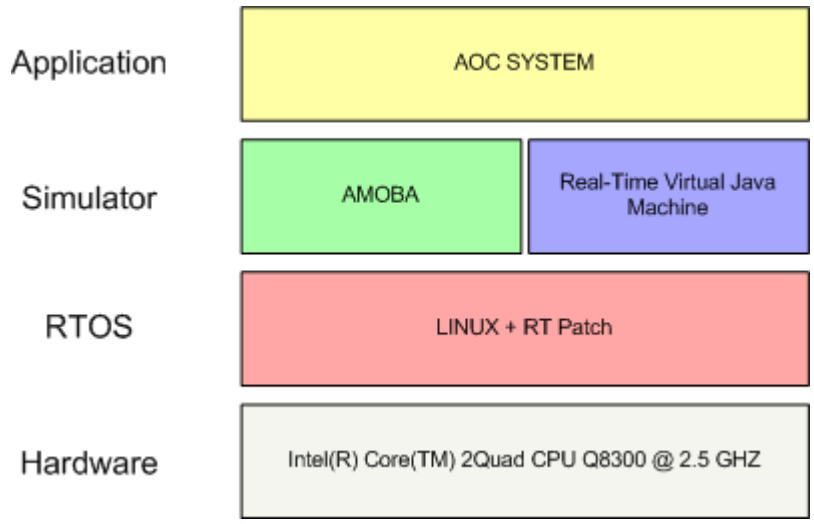

Figure 13– Java AOC Architecture

The inclusion of the real-time virtual Java machine has its sole purpose in to allow the Java AOC partition to run with hard real-time capabilities.

4.3 – Parallel Model for Airline Operational Centre Application

This section describes the parallel model of the AOC system, this model features three AOC partitions working in parallel.

4.3.1 – Parallel Airline Operational Centre Application

The idea behind this model is to find a parallel solution for the AOC system. The proposed model consists of three AOC partitions outside the scope of the original partition scheme, see figure 14.

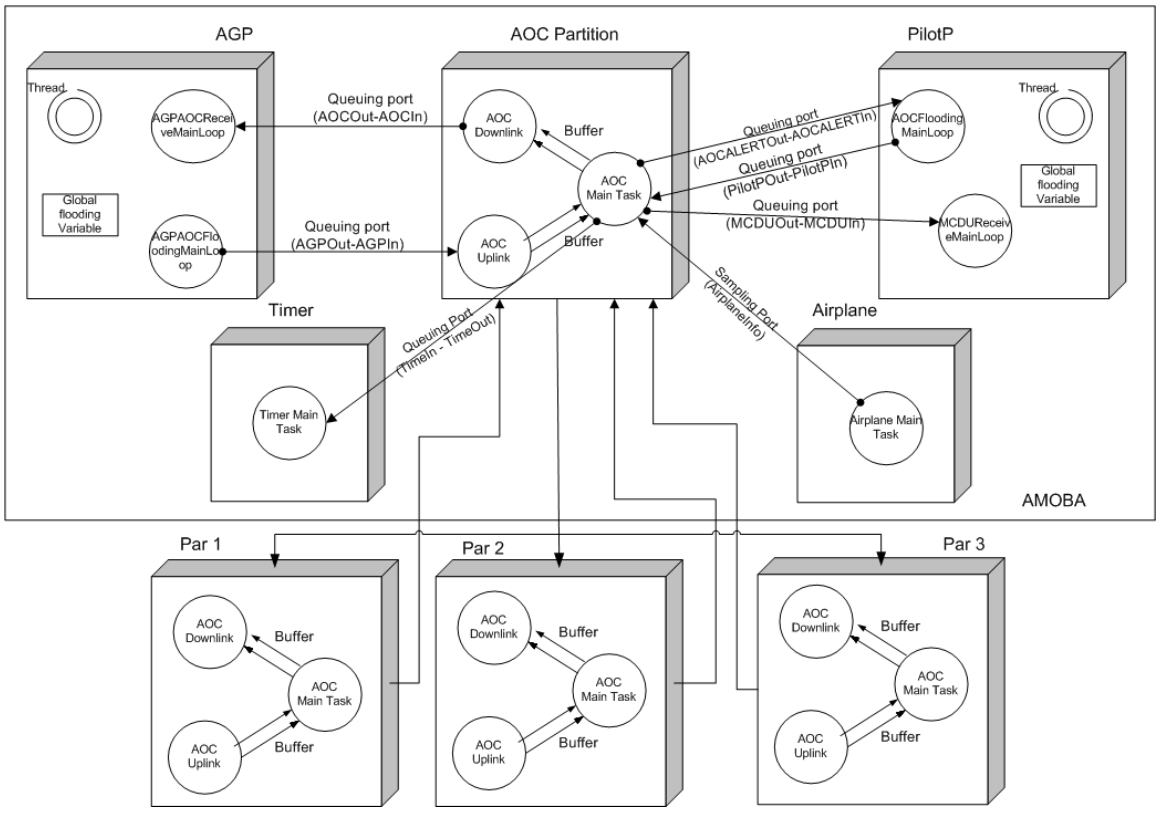

Figure 14– Parallel AOC system

For this model the partitions outside AMOBA are programmed using C instead of Java. For this reason a fourth theoretical model, the parallel Java model, will be discussed on section 4.4.4 with basis of the values found in the Java and Parallel model.

Like the previous models this one maintains the five partitions model scheme under the scope of the AMOBA and adds three AOC partitions outside the AMOBA (Par 1, Par 2 and Par 3 in Figure 14) that will do the message processing. The AOC partition inside AMOBA will much like in the Java model serve to relay the messages received to the AOC Partitions, but unlike the Java model it will also distribute the load between the partitions. In the case of the requests that the pilot makes, the AOC partition must also be able to know in which partition has stored the desired report and relay the request to the proper partition.

Like in the two previous models all the features that require to be maintained are present. The AOC system that falls under the scope of the AMOBA runs with a bind to a single core, while every AOC partition outside the AMOBA runs on its own core. This guarantees that true parallelism is achieved.

This model adds tremendous overhead to the AOC partition in the AMOBA, the sheer volume of messages that it receives and sends is much larger then in the original system, due to the fact that it must send the messages to the partitions outside the AMOBA and receive from those partitions. Also adding to the overhead is the fact that the partition has to know in which AOC partition is stored the requested message, together with the load balancing algorithm. The algorithm although simple must be applied to every message the partition receives from the AGP and Pilot, and that takes its toll.

4.3.2 – Parallel Airline Operational Centre Architecture

In this model all the partitions are made using C programming language and all use the APEX interface. This fact allows this model to have the exact same architecture as the original one, see figure 11.

4.4 – Explanation of the study and Results

In this section the results for each of the three models will be presented together with a discussion of the presented values. Some facts about the expected values applied to the application will also be presented.

4.4.1 – Facts

There are some facts that are to be expected even without the knowledge of the values that will be presented below.

The Java model is expected to be slower then original model, the Java AOC partition is not in any way optimized due to the fact that is in every way possible a match to the original AOC partition. Such a lack of optimization is visible for example by the extensive use of the keyword “new” in the Java model, and the presence of a Garbage Collection also adds to the visible expected slower results. These reasons cause the Java model to be able to process less messages in the same amount of time as the the original AOC application and the parallel model AOC application.

Java is known for having a bit of a slow start, the need to load classes in the beginning of the application causes an accentuated decrease in the Java AOC partition performance. For this reason the first values produced by the partition are values to be discarded, to place all the models on equal footing all the models discard the first 100 messages.

The Parallel model is expected to be able to process more messages than the original version. For the tests a Quad-Core processor was used, but this does not mean that we will have a 400% gain. In order to gain a 400% processing increase, when compared with the original system, in a Quad-core processor we would need a system that would present a extremely simple parallelizing solution, alas this is not the case.

The original and parallel versions will have the same processing time (average, worst and best) for the reports, this is so due to the fact that the partitions of the parallel model are copies of the original AOC partition.

4.4.2 - Benchmarking procedure

For the benchmarking it is important that all the models are placed in the same conditions. This means that the flow of requests and responses by the Pilot and AGP is the same for all the models. In order to obtain visible results in terms of number of reports each model can process in a given time it is necessary to feed the AOC partition more reports then it can actually handle, forcing the partition to reach overflow. This forces the AOC partition (partitions in the parallel model) to achieve maximum processing capacity, and thus process the maximum number of reports it can.

All the models have the same underlying software and hardware support.

Each test took 10 minutes, and several tests were made to each of the models, this way we ensure that the values that are presented below are not an anomalies.

The values that are considered to be interesting as stated in section 4.1.1 are:

1. Average processing time per report
2. Worst case processing time per report
3. Minimum processing time per report
4. Maximal number of reports processed within a given time period

4.4.3 – Results

The results that each of the models produced will be presented in this section, according to the following order the original AOC, the Java AOC and the parallel AOC. The presented values are the values discussed in Table 1.

Original AOC Results

The results that the original model produced are in table 3.

Type	Average Time(μs)	Maximum Time(μs)	Minimum Time(μs)	Total Messages
Create Free Text Downlink Message	**6**	**13**	**6**	**592**
Create Malfunction Message	**16**	**29**	**16**	**592**
Create Load sheet Acknowledgment.	**8**	**22**	**8**	**592**
Create Meteo Request	**8**	**20**	**8**	**593**
Process Periodic Functionality	**21**	**87**	**20**	**592**
Process Pilot Request	**38**	**78**	**29**	**592**
Process Pilot Response	**21**	**53**	**16**	**2370**
Process Uplink Message	**7**	**88**	**5**	**3554**
Average/Total	**15.625**	**48.75**	**13.5**	**9477**

Table 3– Original AOC Results

(10 minutes)

These are the values that will serve for comparison for the other two models. For the Java model the most interesting values to compare will be average, maximum and minimum time for all of the events, the number of reports processed in the same time is proportional to the times of the events. For the parallel model the interesting value is the number of events it can process in the same time as the original application.

Java Results

The results that the Java model produced are in table 4.

Type	Average Time(µs)	Maximum Time(µs)	Minimum Time(µs)	Total Messages
Create Free Text Downlink Message	**31**	**96**	**18**	**194**
Create Malfunction Message	**54**	**97**	**38**	**195**
Create Load sheet Acknowledgment.	**36**	**117**	**22**	**202**
Create Meteo Request	**40**	**146**	**24**	**198**
Process Periodic Functionality	**70**	**127**	**50**	**149**
Process Pilot Request	**93**	**224**	**69**	**176**
Process Pilot Response	**91**	**220**	**54**	**748**
Process Uplink Message	**66**	**462**	**36**	**276**
Average/Total	**60.125**	**186.125**	**38.875**	**2138**

Table 4– Java AOC Results

As expected the Java model proved to be much slower in message processing then the original application, and as a consequence the number of events processed in the same time is drastically smaller see table 4.

It is relevant to mention that the hard Real-Time component of the model was left intact.

In the average time the Java model is 3.848 times slower then the original model.

Java Average Time = 60.125 µs

Original Average Time = 15.625 µs

60.125/15.625 = 3.848 times slower.

In the maximum time the Java model is 3.818 times slower then the original model.

Java Average Time = 186.125 µs

Original Average Time = 48.75 µs

186.125/48.75 = 3.818 times slower.

In the minimum time the Java model is 2.88 times slower then the original model.

Java Average Time = 38.875 µs

Original Average Time = 13.5 µs

38.875/13.5 = 2. 88 times slower.

The Java model in the same amount of time processed 4.43 times less events then the original application

9477/2138 = 4. 43 times less events

Parallel Results

The results that the Parallel model produced are in table 5.

Type	Average Time(µs)	Maximum Time(µs)	Minimum Time(µs)	Total Messages
Create Free Text Downlink Message	**6**	**17**	**5**	**2066**
Create Malfunction Message	**17**	**95**	**15**	**1927**
Create Load sheet Acknowledgment.	**9**	**18**	**8**	**2054**
Create Meteo Request	**9**	**20**	**8**	**1807**
Process Periodic Functionality	**21**	**35**	**20**	**523**
Process Pilot Request	**37**	**81**	**24**	**1547**
Process Pilot Response	**21**	**111**	**15**	**7567**
Process Uplink Message	**9**	**99**	**4**	**5185**
Average/Total	**16.125**	**59.5**	**12.375**	**24814**

Table 5– Parallel AOC Results

As shown in table 5 the average, maximum and minimum times are almost identical as in the original application, but the interesting value is the total number of events that the parallel version processes. While the parallel model processed 24814 events the original model processed 9477. This shows that the parallel model processes 2.618 times more events then the original version.

24814/9477 = 2.618 times more events

As expected this model indeed processed more events when compared to the original model, despite the overhead (mentioned in section 4.3.1), also as expected we did not processed four times more events but instead 2.618 times more events. This is the result of the added overhead of the entire system as well as the nature of the system, like the fact that it is an ARINC 653 application. It is relevant to mention that the hard Rea-Time component of the model was left intact.

4.4.4 – Result discussion

As the results clearly show there is actual gain to be had in the potential of the multi-core architecture. The parallel system vs. the original application showed a 2.618 times gain in speed. The Java model vs. the original application showed in a 4.43 times in speed decline in processing reports in the same amount of time. These values tells us two things, the first is that Java is much slower than C. This can be explained by the lack of optimization of the Java code and the chosen implementation of the garbage collection [35]. The other is that parallelization works and presents satisfactory results. If fact 2.618 is the result obtained with the proposed scheme for the application, there is room for optimization.

The exercise of imagining a parallel Java application with these two values is not hard, if the Java model is 4.43 times slower and the parallel model is 2.618 times faster a simple calculation gives us the Parallel Java model theoretical values:

4.43 – 2.618 = 1.812 times slower.

This value shows that despite the parallelization Java is still slower, of course that this value is applied only to this particular application and to these particular circumstances, and that the Garbage collection implementation may not be ideal for these performance issues. Yet this value allows us to see that it is possible to plot a course to enable some day Java to achieve similar or better results than the more traditional languages.

Chapter conclusion

With an adapted AOC application it was possible to make three different models, all with different characteristics and expectancies regarding their performance. With the results produced by the normal, Java and parallel models it is possible to see the processing capabilities between the models. The differences between the models points towards a slow Java model and a highly capable parallel model. In the next chapter these results will be further discussed.

Chapter 5 – Conclusions and Future Work

This dissertation dealt with the issues related to the implementation of Real-Time Java applications in multi-core environment. Even though several tests have been made to test the differences between parallel/sequential and Java/C, this dissertation intended to show these tests simultaneously and applied to an actual Real-Time application that utilizes the ARINC 653 standard.

Multi-core has become state of the art when it comes to processors, it is a simple exercise to realize that the evolution that processors have been having in the last decades is exponential. Today single core processors are becoming obsolete, being replaced by processors with ever increasing number of cores. One of the great promoters of this evolution is the limitations of the single-core processors.

Java is without a doubt one of the most popular languages today, despite this fact Java until recently had no place in the Real-Time world. With the creation of the RTSJ and SCJT technologies that contained ideas for the implementation of Real-Time Java Virtual Machines, Java found its way to Real-Time.

This dissertation proved that parallelization presents satisfactory results, and even if using Java, the result although slower then the original application, shows a promising path toward Java Real-Time applications in multi-core architectures.

Future work regarding this dissertation includes the implementation of a more suited version that ill be optimized with attention regarding the differences between Java and C. With Java commencing to establish itself in the Real-Time community it's only natural that several Real-Time Java virtual machines come to be. As such, the logical step begins with testing the proposed solution of the application on different Real-Time Java virtual machines.

References

[1] Gordon E. Moore "Cramming more components onto integrated circuits" Electronics Magazine, April 1965

[2] Manek. "Moore's Law is dead, says Gordon Moore". Techworld, April 2005

[3] R.M. Ramanathan "Intel® Multi-Core Processors: Making the Move to Quad-Core and Beyond", 2006

[4] The Real-Time for Java Expert Group, "Real-Time Specification for Java", 2000

[5] Safety Critical Expert Group "Draft - Safety Critical Java Proposal", 25 May 2006

[6] Michael H. Dawson, "Challenges in Implementing the Real-Time Specification for Java (RTSJ) in a Commercial Real-Time Java Virtual Machine", 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing (ISORC), Orlando, Florida, USA, 5-7 May 2008

[7] Vlad Olaru, Anca Hangan, Gheroghe Sebestyen-Pal and Gavril Saplacan. Real-Time Java and Multi-core Architectures. Intelligent Computer Communication and Processing, 2008. ICCP 2008. 4th International Conference.

[8] Sérgio Santos, José Rufino, Tobias Schoofs, Cássia Tatibana, and James Windsor. "A Portable ARINC 653 Standard Interface. In The AAIA/IEEE 27th Digital Avionics Systems Conference", 26-30 October 2008, St. Paul, MN, USA, 2008. IEEE Computer Society

[9] Edgar Pascoal and Cássia Tatibana. "AMOBA WP2: AMOBA Simulator Design. Technical report", Skysoft Portugal S.A., June 2008

[10] Tobias Schoofs. "AMOBA D1.1: Study of ARINC 653 Application to the Space Domain. Technical report", Skysoft Portugal S.A., May 2008

[11] Edgar Pascoal, José Rufino, Tobias Schoofs, and James Windsor. "AMOBA – ARINC 653 Simulator for Modular Space-Based Applications." In Proceedings of DASIA 2008 – Data Systems In Aerospace – conference, 27 – 30 May 2008, Palma

de Mallorca, Spain, May 2008. European Space Agency, Communication Production Office.

[12] José Rufino, Sérgio Filipe, Manuel Coutinho, Sérgio Santos, and James Windsor. AIR - ARINC 653 In RTEMS. In Proceedings of DASIA 2007 – Data Systems In Aerospace Conference, 29 May - 1 June 2007, Naples, Italy, August 2007. European Space Agency, Communication Production Office.

[13] Diana Consortium, "Model for the Execution Environment". DIANA-ANX-DC-2.2 , 11-06-2008

[14] - D1.2 Validation Requirements "JEOPARD - Java Environment for Parallel Real-time Development" June 2008

[15] www.java.com/en/about/

[16] http://www.rtsj.org/

[17] Gill, S. (1958), "Parallel Programming," The Computer Journal, vol. 1, April, pp. 2-10.

[18] http://www.beowulf.org/

[19]http://software.intel.com/en-us/blogs/2007/03/14/the-multi-core-dilemma-by-patrick-leonard/

[20] Intel "Intel 64 and IA-32 Architectures Optimization Reference Manual" November 2007

[21] Gene M. Amdahl "Validity of the single processor approach to achieving large scale computing capabilities" AFIPS spring joint computer conference, April 1967

[22] Sriram Vangal, et al. "An 80-Tile Sub-100-W TeraFLOPS Processor in 65-nm CMOS," l. IEEE Journal of Solid-State Circuits, Vol. 43, No. 1, Jan 2008

[23] Tong Li, D. Baumberger, D. A. Koufaty, and S. Hahn. Efficient Operating System Scheduling for Performance-Asymmetric Multi-Core Architectures. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC07), November 2007.

[24] M. Annavaram, E. Grochowski, and J. Shen. Mitigating Amdahl's law through EPI throttle. In Proceedings of the 32nd Annual International Symposium on Computer Architecture, June 2005.

[25] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture, December 2003.

[26] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-ISA heterogeneous multicore architectures for multithreaded workload performance. In Proceedings of the 31th Annual International Symposium on Computer Architecture, June 2004.

[27] Airlines Electronic Engineering Committee (AEEC). Avionics Application Software Standard Interface - ARINC Specification 653 - Part 1 - Required Services (Supplement 2). ARINC Inc., March 2006. 1.2

[28] Airlines Electronic Engineering Committee (AEEC). Avionics Application Software Standard Interface - ARINC Specification 653 - Part 3 - Conformity Test Specification. ARINC Inc., October 2006. 1.3

[29] R. Kaiser. "The PikeOS Concept History and Design", Sysgo whitepaper.

[30] Sysgo Embedded Solutions. "PikeOS", Sysgo Product Datasheet

[31] Karim Yaghmour. "Adaptive Domain Environment for Operating Systems", Opersys inc.

[32] Karim Yaghmour. "Building a Real-Time Operating System on top of the Adaptive Domain Environment for Operating Systems", Opersys inc.

[33] Michael H. Dawson, "Challenges in Implementing the Real-Time Specification for Java (RTSJ) in a Commercial Real-Time Java Virtual Machine" 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing (ISORC)

[34] T. Schoofs, S. Santos, C. Tatibana, J. Anjos, J. Rufino and J, Windsor, An Integrated Modular Avionics Development Environment – In Proceedings of the DASIA 2009 – DAta Systems In Aerospace Conference, EUROSPACE, Istanbul, Turkey. May 2009.

[35] Dr. Fridtjof Siebert "Limits of Parallel Marking Garbage Collection" International Symposium on Memory Management - June 2008

[36] aicas GmbH "The Virtual Machine for Real-time and Embedded Systems" JamaicaVM 3.2 — User Documentation, April 2008

[37] Radio Technical Commission for Aeronautics (RTCA). Integrated Modular Avionics (IMA) "Development Guidance and Certification Considerations – Document DO-297. RTCA Inc.", August 2005.

[38] John Rushby "Partitioning for Safety and Security: Requirements, Mechanisms, and Assurance. NASA Langley Research Center, Contractor Report (Also issud by the FAA) CR-1999-209347", Computer Science Laboratory, SRI International, June 1999

[39] Airlines Electronic Committee (AEEC). "Avionics Application Software Standard Interface – ARINC Specification 653 – Part 1 – Required Services (Supplement 2)". ARINC Inc. October 2006

[40] Airlines Electronic Engineering Committee (AEEC). "Avionics Application Software Standard Interface – ARINC Specification 653 – Part 2 – Extended Services." ARINC Inc. June 2007

[41] Airlines Electronic Engineering Committee (AEEC). "Avionics Application Software Standard Interface – ARINC Specification 653 – Part 2 – Conformity Test Specification." ARINC Inc. October 2006

[42] Nuno Diniz and José Rufino. "ARINC 653 in Space. In Proceeding of DASIA 2005" – Data Systems In Aerospace Conference, 30 May – 2 June 2005, Edinburgh, UK. European Space Agency, Publications Division. August 2005

[43] Flynn, M., Some Computer Organizations and Their Effectiveness, IEEE Trans. Comput., Vol. C-21, pp. 948, 1972.

ANEX

Work Planning and Thesis Timeline

1. Application analysis concerning optimization to a multi-core platform

01/09/08 – 30/10/08

1.1 Definition of a simulation scope.

01/09/08 – 15/09/08

1.1. Application analysis and definition of the parallel design

15/09/08–31/10/08

2. Java Object oriented Re-design of the application

03/11/08 – 31/12/08

2.2. Re-design of the application in java

03/11/08 – 15/12/08

2.1. Production of the necessary documentation

15/12/08 – 31/12/08

3. Re-programming of the application in java

02/01/09 – 20/02/09

4. Application validation

23/02/09 – 17/04/09

4.1. Validation against Requirements

23/02/09 – 13/03/09

4.2. Bench Marks

16/03/09 – 17/04/09

5. Activity documentation

20/04/09 – 30/05/09

6. Dissertation

01/06/09 – 31/07/09

General definitions and concepts

APEX ARINC 653 defines an APplication EXecutive (APEX) for space and time partitioning. Each partition has its own memory space. It also has a dedicated time slot allocated by the APEX. Within each Partition multitasking is allowed.

Jamaica Virtual Machine is a Java VM (Virtual Machine) for real-time systems, designed to run under hard real-time conditions on real-time platforms. It features deterministic and efficient garbage collection despite the difficulties inherent to its implementation [35], priority inheritance and other necessary real-time functionality. Jamaica VM also supports many of the Java standard libraries and the Real-Time Specification for Java [36].

POSIX Portable Operating System Interface is the collective name of a family of related standards specified by the IEEE to define the application programming interface (API), along with shell and utilities interfaces for software compatible with variants of the Unix operating system, although the standard can apply to any operating system.

Integrated Modular Avionics (IMA) is a shared set of flexible, reusable, and interoperable hardware and software resources that, when integrated, form a platform that provides services, designed and verified to a defined set of safety and performance requirements, to host application performing aircraft functions [37].

The IMA concept proposes an integrated architecture with application software portable across an assembly of common hardware modules. A IMA architecture imposes multiple requirements on the underlying Operating System [38].

ARINC 653 (Avionics Application Standard Software Interface) is a software specification for space and time partitioning. It defines an API for software of avionics [39] that will be extended [40, 41], following the architecture of Integrated Modular Avionics (IMA). It is part of ARINC 600-Series Standards for Digital Aircraft & Flight Simulators [42].

Acronyms

AGP	AOC Ground Platform
AIDA	Architecture for Independent Distributed Avionics
AIR	ARINC 653 in RTEMS
AMOBA	ARINC 653 simulator for modular space-based application
AOC	Airline Operational Centre
APEX	Application Executive
API	Application Programming Interface
ATM	Air Traffic Management
CPU	Central Processing Unit
DASIA	DAta Systems In Aerospace
DIANA	Distributed Equipment Independent Environment for Advanced Avionic Applications
ESA	European Space Agency
FCUL	Faculdade de Ciências da Universidade de Lisboa
FIFO	First In First Out
GC	Garbage Collection
GUI	Graphical user interface
HAL	Hardware Abstraction Layer
IEEE	Institute of Electrical and Electronics Engineers
IMA	Integrated Modular Avionics
JIT	Just-In-Time compilation
JEOPARD	Java Environment for Parallel Real-Time Development
JSR	Java Specification Request
JVM	Java Virtual Machine
MA-AFAS	More Autonomous Aircraft in the Future Air Traffic Management

	System
NHRTT	No Heap Real-Time Thread
OO	Object Oriented
OS	Operating System
POSIX	Portable Operating System Interface
RTEMS	Real-Time Executive for Multiprocessor Systems
RTOS	Real-Time Operating System
RTSJ	Real-Time Specification for Java
RTT	Real-Time Thread
SMP	Symmetric Multiprocessing
SCJT	Safety Critical Java Technology
VM	Virtual Machine