Industry Use Cases for the Java Environment for Parallel Realtime Development

Frederic Lamy
Cassidian Electronics, an EADS Company
Woerthstrasse 85
89077 Ulm, Germany
frederic.lamy@cassidian.com

Tobias Schoofs
GMV, Portugal
Av. D. João II, Lote 1.17.02
Torre Fernão Magalhães - 7°
1998-025 Lisboa, Portugal
tobias.schoofs@gmv.com

ABSTRACT

Multicore systems have become standard on desktop computers today and current operating systems and software development tools provide means to actually use the additional computational power efficiently. A more fundamental change, however, is required to fully exploit the power of multicore systems. Furthermore, the fast growing market of embedded systems, up to now, is much less affected by the introduction of parallel technologies. This is already changing quickly today. Tools for efficient development of reliable embedded software are demanded. This is in particular true for tools that are able to guarantee hard real-time requirements.

The JEOPARD project has addressed these demands by developing software tools to exploit multicore power while ensuring correctness and predictable timing.

In this paper, we will present the industrial use cases that have been developed in the scope of JEOPARD to validate the tools against requirements from real-world industry examples. The first use case is a radar application developed at Cassidian in Ulm, Germany, where hard real-time requirements and demanding performance requirements had to be met; the second is an avionics application developed at GMV in Lisbon, Portugal, where hard real-time requirements and strict safety constraints had to be addressed.

General Terms

Software Engineering, Design, Reliability, Experimentation, Standardization, Verification, Performance

Keywords

Java, RTSJ, safety-critical real-time systems, radar processing, avionics

JTRES 2011 September 26-28, 2011, York, UK

1. INTRODUCTION

Increasing demands are made on embedded systems and greater reliance will be placed on the delivery of their services. More and more of these systems will become high-integrity systems whose failure can cause loss of life, environmental harm, or significant financial loss. The increasing demand on the processing power for the ever growing complexity of services delivered by embedded systems can only be met by the use of multicore systems. In addition to desktop systems, embedded systems have requirements on predictable timing behaviour and safety-criticality.

The JEOPARD project has addressed these challenges by implementing tools for development and execution of Real-Time Java applications running on multicore hardware. The project integrated an execution environment (a real-time operating system (RTOS) and a real-time Java VM) as well as analysis, verification and debugging tools.

The JEOPARD project has been conducted by a consortium of research institutes (Forschungszentrum Informatik (FZI), Karlsruhe, Germany, the Technical University at Cluj-Napoca, Romenia the University of York, UK, the Technical University of Vienna, Austria and the Technical University of Denmark, Copenhagen), tool vendors (aicas, Karlsruhe, Germany and SYSGO, Mainz, Germany) and embedded system developers (RadioLabs, Rome, Italy, Cassidian, Ulm, Germany and GMV, Lisbon, Portugal) and was led by The Open Group. The project was funded by the European Commission through the 7^{th} Framework Programme.

The main strategic objective of the JEOPARD project was to provide the tools for platform-independent development of predictable systems that make use of SMP multicore platforms [**?**, **?**]. These tools will enhance the software productivity and by extending technology that is established on desktop system by the specific needs of multicore embedded systems.

An important part of the project was dedicated to the evaluation of the proposed tools by means of real-world industrial applications from different domains. From these experience, lessons learnt were drawn to better understand the needs of the embedded systems industry.

The rest of the paper is organised as follows. In section 2, we will provide a short overview of the JEOPARD platform. In section 3, we will define the use case objectives. In sections 4 and 5, we will discuss the use cases in more detail. Both sections will present the lessons learnt during the experiments. We will focus on those findings that are related to the challenges addressed by the JEOPARD project, *i.e.* use of multicore for embedded systems. Finally, in section 6, we draw some conclusions.

2. OVERVIEW OF THE JEOPARD PLATFORM

The JEOPARD platform addresses the challenges raised by multicore platforms on different layers:

- The CPU Architecture Layer

  The basis of a multicore architecture is a Chip Multiprocessor (CMP). The JEOPARD project considered three basic variants of a CMP to support Java programs targeted at multicore platforms:

  1. SMP a symmetric arrangement of conventional processors, as typified by Pentium class multicore CPUs.
  2. Multi-JOP – a number of hardware Java processors with some shared memory (based upon the Java processor JOP [**?**, **?**]).
  3. NUMA – to incorporate future architectures, including those with substantial field-programmable elements.

  For all architectures, the project developed efficient support for higher level OS and Java based services. In particular, specific services for synchronisation, scheduling support and deterministic automatic memory management.

  Whereas the SMP architecture is addressed by all layers, including OS and the VM layer, the multi-JOP architecture does not need these layers. Java programs are executed directly on the JOP hardware without the need for an OS or VM.

- The OS Layer

  The OS layer implements resource allocation mechanisms used by applications either at run time or at design time. In the JEOPARD platform, the OS hosts higher level run-time environments, such as Java VMs. It, thus, strengthens the idea of virtualisation and the independence between hardware and software as one of the key ideas in modern software development.

  One of the consequences is that the OS shall provide mechanisms that enable the higher level environments to implement resource usage policies. A good example for this distinction is memory management. The OS provides memory protection mechanisms that are based on resource containers assigned to applications, such that an application cannot access memory that is not part of its resource container. How memory is used and maintained inside an application, however, is not part of the mechanism implemented by the OS. This is implemented by the VM, following, for instance, the Java Memory Model.

  In the JEOPARD platform, resource containers are protection domains which will be called *partitions* throughout this paper. Robust Partitioning is a well known concept for safety and security-critical systems. In avionics, for example, partitioning is used to safely execute a set of real-time applications on the same processing hardware. This is part of an architectural paradigm known as Integrated Modular Avionics (IMA) [**?**, **?**]. In the defense domain, the concept is used to ensure security and known as Multiple Independent Levels of Security (MILS) [**?**]. In JEOPARD, a partitioning OS is used mainly to address needs of safety-critical applications.

  On the OS-level, processor cores can be assigned to applications in two ways, either on partition level or on task (thread) level. The assignment is expressed in terms of *affinity* either during run-time (for threads only) or during design-time. Affinity is derived from the de-facto standard usage in the GNU/Linux OS. Currently however, affinity is not part of relevant standards, such as POSIX or, for IMA, ARINC 653 [**?**, **?**]. Even worse, for ARINC 653, multicore systems appear not to exist. Nothing is specified describing how to configure or program the distribution of applications on processors. Moreover, it is explicitly forbidden to distribute one application over a set of processor cores. This will be discussed in more detail in section 5.

- The VM Layer

  For real-time Java applications to run on multicore embedded systems, a multicore real-time Java VM implementation is required. The runtime services of the VM such as the interpreter, thread synchronisation, memory management, *etc.* must be implemented to exploit the available processors while executing predictably *and* efficiently.

  The biggest challenge for such a Java VM is to provide real-time garbage collection that can execute in parallel and without preemption of the Java application. A non-parallel blocking GC cannot be used in a real-time system. What is needed is at least a parallel concurrent GC that uses a set of CPUs reserved for garbage collection work. But the highest flexibility can be reached only with an incremental parallel GC.

  Such a real-time garbage collector must ensure that sufficient free memory is available even though many processors may be performing memory allocations and garbage collection in parallel. Fine-grained synchronisation techniques need to be developed to ensure that this parallel execution does not suffer from contention. In addition to the implementation of a parallel real-time garbage collector, a theoretical analysis of the level of parallelism that can be achieved is needed [**?**].

  The parallel execution of thread synchronisation via Java monitors requires a careful implementation of the primitive monitor operations such as entering, exiting and notifying. Also, the Java thread scheduler must know about multiple processors, parallel Java interpretation and Just-in-Time compilation.

- The API Layer

  For a standard desktop computing environment, the application programmer is content to allow the OS to manage the access to the computing resources. In an embedded environment this is generally not the case. Better real-time performance can often be obtained by partitioning the applications threads between the core processors. Currently, there is no easy way to do this using Java or the RTSJ API. This is an accepted limitation of the current real-time Java technology. Although some consideration has been given to an API

to do this within the context of JSR 282, the proposal is not based on practical experience [?, ?].

The work on the API layer was concerned with extending the Java programming model to allow access to the resources in a machine-independent way. The focus is on providing support for static partitioning but dynamic aspects were also considered. In addition, the APIs to allow hardware-implemented FPGA components to be integrated within the overall Java framework was addressed.

- The Tools Layer

  Independent of the implementation and run-time aspects, reliable applications running on parallel systems require in-depth analysis of the correctness of the implementation and of time-related aspects such as schedulability, lack of deadlocks, race conditions, *etc.* With the use of multicore systems, parallel execution becomes the norm such that errors that do not manifest as faults on single CPU systems are much more likely to cause system failure. For the correctness analysis of parallel applications, existing analysis tools need to be enhanced to find errors related to parallel execution through static analysis or concurrent unit tests. JEOPARD addressed this challenge by providing tools such as a scheduling analyser based on contracts, a static analyser of Java bytecode to detect deadlocks and race conditions, a concurrent unit testing tool [?] that reduces false positives in deadlock and race condition detection and a thread monitor that helps to understand the actual parallel execution of the code.

3. USE CASE OBJECTIVES

The objectives of the use case are threefold:

- Evaluation of the tools developed in the scope of the project in terms of correctness, timeliness and performance;
- Contrasting the concepts developed in the scope of the project with real-world requirements and a real-world development process. Are the concepts sufficient in terms of design, integration, optimisation and verification?
- Demonstrate the feasibility of the new concepts, in particular the use of Java and the use of multicore architectures, in the context of the industry domains, including safety-critical applications and applications with high performance requirements.

To address these objectives, requirements for the JEOPARD platform driven by the industrial use cases have been defined at the beginning of the project. These requirements included technical means, *e.g.* networking, inter-process communication and real-time constraints the platform shall be able to guarantee. The requirements were derived by real application requirements, such that the evaluation of the applications according to already existing requirements and validation test-beds could eventually be used to evaluate the platform hosting the applications.

A lot of effort has been spent on the first objective, the evaluation of the tools; both demonstration activities eventually resulted in prototypes that, in principle, could be used as applications in their respective domains. The avionics use case, of course, would have to undergo an expensive certification process according to air-safety regulations. The radar use case, on the other hand, would have to undergo further testing to prove its reliability in a demanding application environment.

The most important objective was, of course, the second one, aiming at the evaluation of the concepts proposed and developed in the scope of the project. The results related to this objective lead not only to an enhancement of implementations, such as performance optimisation or bug-fixes, but is fed back into research to review former results and introduce new topics. The discussion of the results of this objective is the main part of the *lessons learnt* at the end of the following demonstrator sections and as such the main result of the JEOPARD industry experiments.

Finally, the third objective is more related to the specific industry domains. The idea is that a successful demonstration of technologies like Java and multicore, may provide evidence to the industries that these new concepts – new in the domain of embedded systems development – are sufficiently mature and reliable to be used in the the respective domains. For the radar use case, Cassidian actually opts for building a new product on top of the JEOPARD prototype; for the avionics use case, GMV extends research already performed in the aviation industry concerning Java and multicore [?].

4. THE RADAR APPLICATION

Figure ?? depicts the main Radar processor components of a typical Air Traffic Control radar. The Signal Processor digitises the received radar echoes, eliminates unwanted detections (ground reflections, weather fronts...) and provides range, azimuth and amplitude of flying objects bundled in detection reports to the clustering process (Parameter Extractor). The Parameter Extractor merges the detections belonging to one target together into a plot to be sent to the tracking process (Tracker). The tracker analyses the incoming plots and uses them to update the tracks sent to the radar display. The Radar Controller dispatches parameters and control information to these three segments and monitors their status.

Cassidian produced two demonstrators running on top of JEOPARD tools.

- Java Tracker (jTracker)
- Java Radar Signal Processor (jRSP)

Although these demonstrators are only prototypes for the sake of evaluation of the JEOPARD tools, they are very similar to the original Radar Signal Processor and the Tracker applications running in several Cassidian products. The main difference between the Avionics and Radar Use Case is the raw computing power requested by the radar demonstrators jRSP and jTracker. On one hand, jRSP processes input datastreams with rates of about 10MBytes/s; these data rates are rising up to 40MBytes/s within the jRSP application. On the other hand, jTracker is confronted only with low data rate inputs, but must keep track of many small objects, each object requesting a considerable amount of computations.

The increasing availability of multicore microprocessors in the embedded field, allows radar processor designers to build more compact systems: one single board computer (SBC)

fitted with a quad-core CPU replaces four to five SBCs fitted with one single core CPU. The JEOPARD tool chain brings to radar applications the ease of using Java and the guarantees for execution predictability in a parallel execution environment. Moreover, JEOPARD provides a practical answer for the need in radar systems to interface microprocessors to non CPU systems via FPGAs thanks to the JEOPARD Tools Hardware Methods and JOP. JEOPARD is therefore a key tool chain fostering the integration of heterogeneous components.

One key achievement of JEOPARD is a working build and execution tool chain with which Cassidian was able to demonstrate a stable jRSP at the European Commission's ICT event, September 2010 in Brussels. Moreover, the JEOPARD debugging and analysis tools ThreadMonitor, Concurrent Unit Testing (*cJunit*) and *Veriflux* provided decisive support for the debugging of the parallel applications.

The JEOPARD evaluation benchmarked the JEOPARD real-time execution environment against the non real-time Java Virtual Machine (*i.e.* OpenJDK) and Operating System (*i.e.* Linux). The main goal was to measure the gain brought by the JEOPARD tools in terms of reproducibility and evaluate the expected loss of average execution speed against the non real-time mainstream tool chain. The main evaluation criteria are:

- best, average and worst execution time
- scalability (parallelisation gain)
- Interface communication overhead

Using Java enabled us to use the same code with only minimal or no changes and run it on different platforms:

- OpenJDK on Linux
- Aicas Jamaica Java Virtual Machine on Linux with real-time scheduler
- Aicas Jamaica Java Virtual Machine on PikeOS

4.1 Java Tracker

In an Air Traffic Control radar, the tracker maintains already confirmed tracks and initiates new tracks using the target information updates provided by the clustering application. The tracker:

- processes incoming plots from the clustering application and incoming messages from the radar controller
- associates plots with existing tracks
- updates, initiates or deletes tracks
- outputs track updates to the radar display

The jTracker test scenario is produced off-line and produces plot data describing 600 synthetic targets. This scenario is putting a relatively high load on the tracker and generates higher traffic than standard ATC radars are used to cope with. This scenario provides a realistic load for the JEOPARD evaluation.

4.1.1 jTracker results

For the average case, execution times of jTracker were disappointing. The average execution times were 2 to 5 times slower when using the JEOPARD tools than running on the combination OpenJDK/Linux. The much more important worst case execution time, however, was very good. The JEOPARD tools cut the maximum latency to the halve of that of the OpenJDK/Linux platform.

The layered software architecture of the Java execution environment enabled us to evaluate with minimal changes the behaviour of jTracker running on the Jamaica Parallel Java VM and standard OpenJDK VM either on Linux or on PikeOS. This will enable us to adapt our products to different customer requirements much faster than today: use of Linux and standard Java VM if criticality is low and price is an issue, on one hand, use of Jamaica Parallel VM and PikeOS for safety-critical applications on the other hand. At the time of writing, Cassidian is planning the adoption of the Jamaica Parallel VM in future tracking products.

4.2 jRSP

4.2.1 Application Overview Corrected

jRSP processes real-time simulated data provided by the radar scenario generator and outputs detection reports to a radar display (PPI) as depicted in figure **??**. The radar scenario generator is FPGA based and provides the same data (same format and timing) as the front end (analogue to digital converter) in the real radar system.

4.2.2 jRSP Architecture

- how the user is involved by this demonstrator: the user can actually see the results of radar processing displayed on the PPI and task from the PPI the execution of the high resolution spectrum estimation algorithm HRSE as well.
- the LCD display, showing the results of HRSE

The use of an LCD display as featured by very simple devices (handheld calculator for example), requires jRSP to provide software and hardware interfacing for such a simple device.

The jRSP architecture (figure **??**) is based on standard high-end PC and FPGA hardware comprising:

- one Intel 6 core CPU i7-980X
- multi PCIe (PCI Express) slot mainboard fitted with 3GB DDR3 RAM
- multiple Gigabit Ethernet interfaces
- two Xilinx ML505 PCIe FPGA cards

The software tools used to run jRSP are:

- a Java Virtual Machine (Jamaica Parallel VM or OpenJDK)
- a Java Virtual Machine for FPGA: Java Optimized Processor (JOP)
- an operating system: PikeOS or Linux
- a software suite seamlessly interfacing Java applications to FPGAs: Hardware Methods (HWMethods)

jRSP receives the data from the scenario generator over Ethernet (UDP). On the multicore CPU, The data is packaged first, then fed through the pulse compression, spectrum estimation and detection. The result of the detection is sent via Ethernet (TCP) to the PPI display. When the user tasks jRSP to perform fine spectrum estimation, the control thread on the multicore CPU receives position information from the PPI display and forwards the required data picked up at the output of the pulse compression to the FPGA running the high resolution spectrum estimation HRSE. Communication between the control thread and HRSE is completely handled by Hardware Methods: FPGA and CPU software developers only „see" a software interface. Hardware Methods creates code frames and handles the transfer of data via PCI Express. The control thread then forwards the result computed by HRSE to the second FPGA in the system running JOP. This last communication scheme is handled by a reduced feature Java RMI running over Hardware Methods [**?**].

4.2.3 jRSP implementation

One great advantage of the JEOPARD tool chain is that without changing the top level design, developers can actually leverage the processing capabilities for their application by parallelising the demanding computations. This is exactly how jRSP was developed: the processing chain was first implemented with a single core development flow and checked for computing correctness. At that stage, performance was nearly irrelevant. Then the application benchmarks showed where the parallelisation was most needed. These benchmarks were made by inserting time measurement points within the code and/or visualising the different threads durations and interactions using the Thread Monitor tool.

Figure **??** show that jRSP threads are linked together via FIFOs and that the pulse compression is run on two parallel threads and the spectrum estimation is run on four parallel threads. Although the first benchmarks were performed without parallelisation, the software architecture of jRSP was from the start designed to run in parallel: simple parameters enable to run the pulse compression and the spectrum estimation either on one single or two, three or four parallel threads.

4.2.4 jRSP results

The JEOPARD evaluation with jRSP shows that the execution times achieved using non real-time tools are 2 to 7 times shorter (better) in average depending on the algorithm and the number of threads running in parallel. However, the JamaicaVM in combination with Linux with real-time scheduling or PikeOS show a smaller and more constant (better) execution time jitter even if the number of cores increases.

With the best parallelisation configuration (two threads for pulse compression and four threads for spectrum estimation), both jRSP realised with JEOPARD tools and non real-time tools can process the input data received over UDP with no loss. Although the CPU load with the JEOPARD tools is higher (60-70 percent) compared to the CPU usage of the non realtime tools (about 20 percent), the JEOPARD tools have the advantage to provide periodic threads (non existent in standard Java) that guarantee that the processing deadline will always be met. jRSP complete application latency results show that the overall application latency requirement could be safely reduced to 14-15 ms if using Jamaica parallel VM instead of OpenJDK. The latency of HWMethods under the three variants of jRSP is relatively small (less than 0.5ms, average 0.2ms) compared to the input burst (frame) rate of jRSP opening new FPGA co-processing options for standard and real-time Java application running on CPUs. At the start of the JEOPARD project, HWMethods was meant to provide FPGA co-processing only for one card. jRSP demonstrates the support of HWMethods for multiple FPGAs which was swiftly added towards the end of the project. JOP was used to interface to the LCD display and relay the results of the high resolution spectrum estimation over HWMethods from the application running on JamaicaVM. A Java RMI functionality for simple objects was implemented for communicating the HRSE results from JamaicaVM to JOP[**?**].

4.3 Lessons Learnt

During the radar use case, the following lessons have been learnt:

- There is little performance gain when running an application on a multicore platform if this application has not been designed to run in parallel from the start.
- Modifying a single core application to run on a multicore platform can take as much or even more time as redesigning the whole application from scratch.
- The overall system performance (execution time) depends highly on a tight integration of the Java VM with the underlying OS. The best performance is achieved by tuning specific parameters of the VM and/or the OS.
- JEOPARD connects CPUs to FPGA seamlessly. This makes the overall platform extremely hardware independent and easily portable.
- For applications demanding all the processing power a multicore CPU can provide, we found useful to „switch off" the garbage collector by allocating memory at program start as object pools.

5. THE AVIONICS APPLICATION

5.1 Java in Avionics

The interest of the avionics community in Java is not new. Java is used as a language for developing host-based tools that are not subject to the rigid certification requirements for on-board systems. Moreover, Java is used to prototype applications that are later completely redesigned and rewritten in a language that is considered more appropriate for the verification strategies applied to critical embedded software, such as C or Ada. In such a prototype, aspects like timeliness and safe memory management are of minor importance. The programmer does not need to cope with the subtleties of worst case execution time or worst case memory allocation.

On the other hand, the process appears to be inefficient. The software is in fact built twice: First as a host-based demonstrator, second as a critical target application. An alternative would use Java not only for prototyping, but would stepwise improve the first implementation to later meet non-functional requirements such as timeliness and memory allocation. The idea is to set up a process where software development would start using full-fledged Java and would move progressively to a realtime version of Java. It is expected that such a process would enable the avionics software

developer to (i) benefit from the features of standard Java to achieve high level of flexibility and responsiveness during the initial development phases, and (ii) benefit from the determinism and safety of realtime versions of Java during the late development phases [**?**].

In the scope of the DIANA project [**?**], Atego's Java realtime environment for safety-critical systems, PERC Pico, was used to implement such a process. PERC Pico differs from other Java platforms in that it does not comprise run-time garbage collection. Instead, the developer annotates the code to describe the scope of the usage of memory objects. Memory objects are kept in a stack from where they are automatically removed according to their scope [**?**].

During the DIANA activities, it turned out that this approach is, on one hand, very interesting for safety-critical software development; the annotations describe the intended memory usage and are, hence, as good as explicit memory management in the code itself with a language such as C for example. It is, on the other hand, often necessary to refactor application code that was initially written without taking memory management into account. It turns out that the literal understanding of the prototype approach sketched above is too naive for application to real world projects. Totally ignoring the memory profile of an application during its design leads to code that is later very difficult to describe in terms of static annotations.

Garbage collection is therefore an interesting aspect of higher-level languages even in avionics. The question is if it is possible to guarantee the run-time behaviour of garbage collection [**?**, **?**]. One of our interests in JEOPARD, was to experiment with a garbage-collecting Java VM and its applicability to critical software development.

5.2 Multicore in Avionics

Multicore hardware will be reality in avionics very soon. The challenges raised with parallel applications are therefore studied by all major software and system developers in the avionics industry.

It seems to be consensus among researchers and practitioners that, in partitioned systems, parallelism must be implemented on the partition level only [**?**]. This means that entire applications, together with their virtual execution environment (partition operating system, language support libraries, I/O access libraries, *etc.*) are assigned to different cores. While this helps to avoid a wide range of errors typically found in parallel systems, and also protects applications from each other, it does not solve the problem of competition for resources of applications executing in parallel. Strategies have to be found to arbiter the parallel access to resources in a predictable and efficient manner. The strategies have to be implemented in a way that allows for defining safe partition scheduling and efficient inter-partition communication.

One of the approaches is to separate resources, usually shared among applications in SMP systems. This leads to a non-uniform memory architecture that can be described as a hardware-based implementation of robust partitioning [**?**]. On one hand, NUMA favours predictability and, thus, safety. On the other hand, it adds requirements which make it even more difficult to benefit form general purpose hardware, for which SMP appears to be the standard architecture today.

In the context of JEOPARD, we studied software development for multicore systems from a different point of view. Even if partitioning is an important aspect of our use case application, we wanted to gather experience with parallelism on task level, *i.e.* we ignored the major trend in avionics research today and redesigned our application to run on multiple CPUs. Our particular use case, the software part of a communication management unit (CMU), benefits indeed from such a design. There are many tasks in the software that, potentially, can run in parallel and can do so in very different ways. We were interested in the impact of parallelism (i) on the run-time behaviour of our application and (ii) on the software development process.

To address the latter, we used most of the tools that were developed in the scope of the JEOPARD project, such as contract-based scheduling analysers, a concurrent unit-testing tool, static analysers and thread monitors. It turned out that parallelism brings, indeed, new challenges that have to be tackled by tools. We were, in particular, confronted with run-time behaviour that differed from our expectations (see section 5.4). It would have been very difficult, if not impossible, to understand the issues behind such surprises – usually locking on shared resources – without the help of the JEOPARD tools.

Additionally, the concurrent unit testing tool in combination with the static analyser appears to be essential for creating the evidence of correctness of parallel software in the scope of a certification process. We implemented a *mini-process* to use these tools efficiently in combination: We used the static analyser to find suspicious code; we then analysed the results to identify false positives. Only those cases for which no good explanation could be found were further investigated using the more labour-intense *cJunit* tool.

Concerning the run time behaviour, we were convinced from the beginning that the use of parallel processor cores would lead to significant speed-up in our application and, hence, to optimised resource usage. Our main objective, however, was not so much the speed-up itself, but to implement a fine-grained control over system resources. We will discuss this aspect in more detail in subsections 5.3 and 5.6.

5.3 The AOC

Our use case was based on the Airlines Operations Centre (AOC). The AOC is the on-board part of a Mission and Trajectory Management (MTM) system. It can be roughly described as a router that manages reports and report requests that are sent between the aircraft and ground systems as well as between different on-board subsystems. It is controlled by the pilot through a Multifunctional Control and Display Unit (MCDU), see figure **??**.

The AOC is with 30 KLOC already a complex embedded application; it was originally written in the C language addressing the ARINC 653 APEX [**?**]. It was developed according to the certification requirements of DO-178B DAL C [**?**] and is, as such, a moderately critical application. The AOC runs with a frequency of 10Hz with deadlines down to 30ms.

For the purpose of our experiments, we ported the application to Java and redesigned it to take advantage of multicore architectures. The application was hosted with the JEOPARD execution environment on a quadcore desktop PC. We used the real-time operating system PikeOS as well as the IMA simulation environment SIMA [**?**] on top of a patched Linux kernel [**?**]. In both configurations, the underlying platform – composed of operating system and execution environment – guarantees hard real-time constraints and enabled us to measure worst case execution times. We used the Linux/SIMA configuration mainly for development purposes, *i.e.* debugging and testing whereas JEOPARD on PikeOS was our intended target platform. Note that this combination of a development and a target platform resembles the prototyping approach, described above in section 5.1, with the important difference that both platforms were real-time systems and, this way, even the development platform allowed for execution time analysis. The prototyping approach practiced in JEOPARD was, hence, not a one-way process as the one described above, but enabled us to go back and forth in iteration cycles.

As with the original application the AOC was hosted together with three other applications on the same computer separated in time and space by the underlying partitioning OS, *i.e.* PikeOS and, respectively, Linux/SIMA. In contrast to the original target, the three other applications are related with testing purposes: Two applications are communication stubs, one connected to an MCDU simulator, the other to a ground system mock-up, both running on a different computer; the third application is a test driver producing data such as ground speed, flight level, cabin temperature, engine temperature, expected time of arrival on a waypoint, *etc.*

5.4 Redesign of the AOC

In the original design, the AOC has three threads, the Downlink which sends messages to client applications and to ground, the Uplink which receives messages from client applications and from ground and the MainLoop which implements the application logic. Figure **??** shows the processing in detail (omitting the Downlink which is not relevant for our purpose).

When a message is received, it is first stored in a tempo-

rary buffer. This buffer is read (at 10Hz) by the MainLoop and the messages are stored in a database. Messages entering the system in this way are reports, for example expected time of arrival on a waypoint (sent from the aircraft to ground), weather reports (sent from ground to the aircraft) or free text messages (exchanged between pilot and ground control). The database is a repository of reports kept in main memory.

When the pilot requests a report, a message is sent to the AOC. As can be seen in the figure, requests are not processed by the Uplink thread but directly by the MainLoop. The MainLoop parses the request and searches the requested report in the database, *e.g.* the latest weather report. When it is found, it is sent back to the pilot.

We developed two parallel designs to optimise this process. The first approach aims at parallelising the heavy message decoding task. To achieve this, we isolated the activity *Process Request* (see figure **??**) in a dedicated thread.

This approach was successful only for best case execution time. The best case went down from 20ms to 5ms. The worst case, however, was extremely bad, so bad that we were not able to meet our deadlines anymore. The worst case went up from 30ms to almost 60ms with two parallel message decoders and to more than 80ms with three parallel decoders.

We identified the problem quickly, using the tools of the JEOPARD platform, in particular the Jamaica Thread Monitor. The issue was that the Java library, we used for the message decoding, shares objects between threads. This, of course, causes the threads to synchronise on these objects and, hence, leads to locking and overall slow-down. To overcome this problem, we rewrote the library to use only local variables.

Still, the approach had a drawback. Since the new thread implements a part of the processing chain of the MainLoop, the two threads have to synchronise each time a message request arrives, again slowing down the overall process. Therefore, we redesigned the application in a more radical fashion. The decoding of messages is part of the MainLoop thread in the original design, in other words messages are stored as text in the database and transformed into report objects only when requested. (Which, of course, has the additional drawback that messages are transformed each time they are requested.) We changed this logic by inserting a new thread on the other side of the database. The message buffer is, hence, not read by the MainLoop anymore, but by the new thread which then would decode the message and store the resulting object in the database.

With this approach, that also led to a more balanced and, indeed, more aesthetic design of the MainLoop processing, we achieved a speed-up of almost 50%. Even more important, we could demonstrate that the resulting AOC scales well to the number of processing cores. When each decoder thread is run on one processor, the number of reports sent in the same time interval can be duplicated by adding one processor core and one decoder thread without the necessity to assign more time resources to the application. This fine-grained control over resource allocation was our main goal for this exercise.

5.5 Porting AOC to Java

The task of porting the application from C to Java was, as expected, smooth and painless. Thanks to Java memory management and available Java libraries, we achieved significant simplifications of the code.

As a general tendency in moving from C to Java, we can point out an abstraction away from low-level concerns to logic representations of architectural patterns, for example representing memory as objects. Here, however, we also encountered a pitfall that manifested itself in the slow-down described in the previous section. The cause of the slow-down was indeed an issue in one of the libraries. It seems paradox that the details of concrete resource usage on a very low level, *e.g.* sharing objects in main memory, instead of using local objects which are free from lock contention, return, when one starts to analyse the time behaviour of code that was developed ignoring those details in the first place.

Under the perspective of the prototype process, described in section 5.1 above, however, this does not appear so paradox anymore. Indeed, in critical software development, the pure development activities are not the main cost factor. What matters more are the activities to provide certification evidence. Even if we start with a full-fledged Java environment drawing on the plentiful resources of available libraries, we are later obliged to either substitute third-party code by own developments with already available certification credits or we have to produce certification evidence for the foreign code. The application, using all the available libraries, is, thus, not the end product, but definitely a mere prototype.

Since the JEOPARD VM provides automatic garbage collection, we did not encounter the difficulties of refactoring the application for being annotated as in the DIANA project. Instead, we hoped to be able to demonstrate formally (i) that the GC would never cause another real-time thread to miss its deadline and (ii) that it would always be able to free sufficient memory so that no thread would run out of memory.

Although, some work in this direction had already been done [**?**], we were disappointed regarding the AOC application. We faced the issue that the application is, due to its internal structure, too complex for a reliable formal analysis – at least in the scope of the JEOPARD project. The point is that time-critical tasks (*e.g.* the decoder thread) allocate memory that is used also by other tasks (*e.g.* the report database or the temporary message buffer). The interdependencies between the threads and the shared memory turned out to be very complex. Note that providing a correctness proof may exceed the effort of using annotations instead. Annotations can, in fact, be seen as a special way to provide the evidence that memory management for a given application is feasible. The difficulties we encountered with a formal demonstration are, hence, related to the same problem that led to the need for refactoring the application in the DIANA project.

The alternative to such a formal proof is testing. It is, however, doubtful if testing alone will be accepted by certification authorities as sufficient evidence for a GC based application. For the time being, GC seems to be limited to applications of lower criticality, like DAL D (minor impact of failure). Our experiments showed on the other hand that critical embedded software reaches a level of complexity that makes the use of higher-level languages with dynamic memory management advantageous.

5.6 Lessons Learnt

During the avionics use case, the following lessons have been learnt:

- Multicore is a good means to enhance control over resource usage. In particular, we can express clear usage domains in terms of processor cores and threads. This is a very satisfying result.
- The parallelisation strategy must be carefully planned. It is, in particular, essential to design memory objects with regard to their usage in different threads that, potentially, execute in parallel.
- Tools and methods are needed to provide certification evidence for parallel applications; the static analyser *Veriflux* and the *cJunit* tool in combination provide a very good starting point. More research is, however, needed to reduce the number of false positives and to support formal reasoning about the code.
- Tools are also needed for debugging parallel applications. The run-time behaviour is sometimes difficult to understand without information on the actual execution of threads. The Jamaica Thread Monitor turned out to be very helpful in this respect.
- Java, as we already knew, is an appropriate language for applications found in the aviation domain. Care must be taken when using the strong abstraction means provided by the language, such as dynamic memory management and standard libraries.
- Our application turned out to be too complex for a formal analysis of correctness with regard to garbage collection. This may limit the use of GC based VMs to applications of lower criticality. More research is needed on this aspect if extremely critical applications are to be addressed in the future.

6. CONCLUSIONS

In this paper, we discussed two of the industrial experiments performed in the scope of the JEOPARD project to evaluate the JEOPARD development and execution platform. The objective of these use cases was to evaluate the maturity of the tools developed during the project, to validate the driving concepts using real industrial requirements to gain evidence that the concepts are able to solve real world problems, on one hand, and to demonstrate their feasibility and, this way, to support their adoption in the industry on the other hand.

In both use cases, improvements of the original software was achieved by using the JEOPARD tools. For Cassidian, one of the main achievements is a platform that can be easily adapted to customers' needs, shortening time-to-market and, hence, establishing a competitive advantage. The means to achieve this is the platform abstraction implemented by Java and the processor core allocation by means of affinity sets provided by the VM and underlying OSes. In particular interesting for Cassidian in this context is the smooth integration of FPGAs into the Java platform using hardware methods.

For the avionics use case, two important achievements can be reported: A fine-grained control over resource usage by means of multicore technology and the establishment of a process based on prototyping functional requirements with Java and integrating non-functional requirements progressively without a tool gap from the host-based prototype to the final target as it is reality today when host prototypes must be re-implemented in another language, such as Ada or C. The use of a GC on the target system appears to be a promising path for less critical systems. For systems with high criticality levels, however, it still appears to be difficult to produce the necessary certification evidence.

The use cases also provided some insight into the challenges raised by multicore systems and the methods and technologies to face them. Both use cases revealed the necessity to carefully design parallel applications. Just adding more processor cores and hoping for positive performance impact is not the way to go. Instead, parallelism must be considered in all stages of the development process. In particular, it appears to be of major importance to reflect the use of memory objects in parallel tasks. In the radar use case, this led to issues during the migration of the application, such that the porting to a parallel version turned out to be more work than it would have been to design the application for multicore in the first place.

Finally, both use cases made intense use of the JEOPARD analysis and debugging tools. Tools, such as the Jamaica Thread Monitor, proved to be helpful if not essential during design and development; tools like the static analyser *Veriflux* and the concurrent unit testing tool *cJunit* are essential for further verification activities.