Finding strategies for executing Ada-code in real-time on Linux using an embedded computer.

ADAM LUNDSTRÖM

KTH Electrical Engineering

Master's Degree Project
Stockholm, Sweden March 2016

MMK 2016:12 MDA 524

Abstract

A combination of Ada, real-time Linux and an embedded computer is a cost-effective solution that accommodates most of the demands of embedded systems development or prototyping. Linux in its standard configuration is not suitable for real-time applications, but there exists solutions that modifies it to be more real-time capable. The question is, what modification is the optimal one from an Ada perspective? To answer this, a literature study has been conducted to identify solutions, followed by an analysis that figures out the most promising one. The selected solution has then been further evaluated and verified by benchmarks and tests running on a BeagleBone Black.

The evaluation results shows that the PREEMPT_RT real-time patches for Linux is the optimal solution for enabling real-time execution of Ada code. Two other promising solutions were Xenomai and RTAI, they provide better performance in terms of lower latencies but does not have full Ada support and requires code to be specifically targeted to their solutions compared to PREEMPT_RT that is transparent to the user.

The worst case latencies for PREEMPT_RT were measured with Cyclictest while the system was stressed by using Sysbench, Hackbench and ping flooding. The tests stressed different part of the system, e.g CPU, memory and file IO, making it possible to determine how sensitive the latencies are to different types of applications. Two of the tests stood out from the others, the ping flood and the Sysbench Thread-test. When pinging the system the worst case latencies were 364 µs, in the order of three times higher than the other loads. The other deviating result was observed when the system was loaded using the Sysbench Thread-test, the latencies were actually lower compared to the unloaded system, 62 µs versus 90 µs. The reason for this is difficult to determine due to the size and complexity of Linux, it would require a deeper analysis of the kernel code.

PREEMPT_RT allows existing applications for Linux to run without modification to the source code which makes it attractive for developing mixed type systems that require real-time predictability, general purpose flexibility and high throughput. It is a cost-effective solution that could be used for teaching Ada and making prototypes that don't require the highest levels of safety certification. The latencies are not low enough to accommodate the demands of all systems, but many systems require latencies only to be in the order of milliseconds, which this solution would be suitable for.

Sammanfattning

En kombination av Ada, realtids-Linux och en enkortsdator är en kostnadseffektiv lösning som möter de flesta av behoven för utveckling och prototypframtagning inom inbyggda system. Linux är i sin standardkonfiguration inte lämplig för realtidsapplikationer, men det finns lösningar som gör Linux mer realtids anpassat. Frågan är, vilken lösning är den optimala från ett Ada perspektiv? För att svara på detta har en litteraturstudie utförts för att identifiera olika lösningar, följt av en analys som tar fram den mest lovande. Den utvalda lösningen har sedan utvärderats och verifierats genom tester som körts på en BeagleBone Black.

Utvärderingen visar att lösningen PREEMPT_RT för Linux är den optimala för realtids-exekvering av kod skriven i Ada. Två andra lovande lösningar är Xenomai och RTAI, de uppvisar bättre prestanda genom kortare fördröjningar. Men de har inte fullt stöd för Ada och kräver att kod anpassas för deras lösning till skillnad från PREEMPT_RT som är transparent för användaren.

Fördröjningarna för PREEMP_RT mättes upp med Cyclictest samtidigt som systemet belastades av Sysbench, Hackbench och 'ping flooding'. Testerna belastade olika delar av systemet, till exempel CPUn, minnet och fil-IO, vilket gör det möjligt att bestämma hur känsligt systemet är för olika typer av applikationer. Två test särskilde sig från de andra, 'ping flooding' och Sysbench Thread-test. När systemet pingades mättes fördröjningarna upp till 364 µs, i storleksordningen tre gånger högre jämfört med de andra testerna. Det andra utmärkande testet var när Sysbench Thread-testet kördes, fördröjningarna var oväntat nog mindre jämfört med det obelastade systemet, 62 µs respektive 90 µs. Anledningen till det är svårt att avgöra på grund av storleken och komplexiteten av Linux, det skulle kräva en djupare analys av Linux-kärnan.

PREEMPT_RT tillåter att befintliga applikation för Linux att köras utan förändringar av källkoden vilket gör lösning attraktiv för utveckling av system som kräver realtidsegenskaper, flexibilitet och hög prestanda. Det är en kostnadseffektiv lösning som kan användas för utbilding i Ada och utveckling av prototyper som inte kräver högsta nivån av säkerhetscertifiering. Fördröjningarna är inte tillräckligt låga för att kunna möta kraven för alla system, men ofta är kraven i storleksordningen av millisekunder, vilket den här lösningen skulle vara lämplig för.

Acknowledgements

I would like to thank my supervisors Xinhai Zhang and Sagar Behere for their guidance.

Contents

1 Introduction

1.1 Motivation - Complex Systems

Self-driving vehicles are lurking in the horizon, robots are moving into our homes and the fusion of sensors can be used for analyzing our mental and physical state. These types of systems can be called embedded, cyber-physical, real-time (see section 2.3), safety-critical and 'Internet of things'. For people familiar with these systems the differences between them might be clear, for others it's different names for the same thing. This thesis is concerned with all these systems and will not divulge into definitions, merely the viewpoint here is that usually these systems are some sort of embedded systems, sometimes connected in a networked fashion, sometimes isolated from other systems and sometimes connected to the Internet. With more than 98% of all the microprocessors being embedded [1] it's a significant area for research.

These systems are becoming more complex. Since cars are a seemingly universal way to explain things examples will be provided from that domain. As [2] explains, the car has evolved through systems of isolated functions to cooperative systems where the functions are connected in networked form. They mention how one of the first systems, like anti-lock breaking, was an isolated function only concerned with its own task, but now there are more advanced systems, for example active break assistance (ABA) that makes connections between several functions like breaking, steering and input/output from the dashboard.

There is a trend towards developing self-driving vehicles, a challenge is how to make them safe. One way to solve the safety issues can be by having a person behind the wheel, ready to intervene in case of computer failure. But if cars evolves in the same way as aircrafts do, being impossible to operate without computer control [3], the computer system must control the vehicle in all situations without relying on a human to intervene if it fails. This full autonomy puts enormous pressure on the software developers and the computational hardware.

With many functions operating at the same time it can be beneficial to assign them with different levels of critically, directing more effort on guaranteeing correct behaviour of the safety-critical ones. It is of essence that the safety-critical functions always works as intended, even if other functions fail. A bug in the infotainment system of a car are not allowed to affect the engine control or breaking system. The traditional approach, known as the federated approach, have been to build these systems by assigning functions to individual electronic control units (ECU) which then provides isolation and protection by hardware

[4]. If a task fails its impact is limited to the ECU it operates on. This approach have the benefit of simplifying the verification and certification process[5], often it is boiled down to bandwidth and timing analysis of the communication bus [6].

The federated approach have for some type of systems reached its limits. The increasing number of functions and their tightly coupled behavior makes sharing of computation and communication resources inevitable [5], also all the wiring and all electrical components for the federated this approach is a significant cost and overall obstacle [6]. A way of providing isolation between functions, but at the same time optimizing resources in terms of weight, power, size and cost is to group functions on fewer amount of ECUs with software provided partitioning, also know as the integrated approach [6, 7]. This has been done in aviation where the previously separated the flight control and autopilot have been integrated on the same computer [3]. The problem of that approach becomes how to guarantee the functions to operate in isolation and to always have the computational power, memory and I/O access that it needs.

Environmental aspects are becoming an all more important factor, the whole life cycle needs to be considered. The benefits of reduced weight and energy consumption for the integrated approach are well suited with today's requirements. Further, it also means that less chemicals and materials used during production. Also, at the end of the life cycle there are less hardware to recycle. The solution presented in this thesis, where a single hardware platform hosts many types of functions, have these benefits.

There is a lot of money and effort to save when developing these systems, the cost associated with making real-time systems to behave as intended is up to 50% of the total cost and even more for systems that are safety-critical [5]. Indeed, for fly-by-wire aircraft manufacturers the validation and certification is so expensive that in order to ensure a 50 year production cycle the manufactures buy a 50 year supply of microprocessors all at once, from the same production lines and masks, to avoid redoing the validation and certification process [8]. But as Lee [8] notes, this also means that no improvements will be done for 50 years, all new technology advancements in efficiency will have to wait, achieving predictability is rendering other factors nearly irrelevant.

The term *complex systems* has been used a couple times so far, it deserves a further explanation. Kopetz view is: *"We classify a system as complex if we are not in the position to develop a set of models of adequate simplicity"*, and further: *"Complexity can only be assigned to models of physical systems, but not to the physical systems themselves, no matter whether these physical systems are natural or man*

made." [5]. This means that complex systems become non-complex if we can simplify our view of them, this absolutely necessary to cope with the challenges: *"There is no alternative to simplicity "* [9]. Kopetz present four ways to achieve simplicity; abstraction, isolation, partitioning and segmentation. A fifth way is mentioned by [10], using hierarchies to reduce the number of objects a human needs to handle. Especially abstraction is echoed throughout the literature as a fundamental concept for dealing with complexity [3, 8, 11–13]. Basically abstraction removes details that are irrelevant for the problems that needs to be solved [5]. At the programming language level this can be done by for example using Ada, a more high-level language than C which is typically used for embedded systems. Below that, more abstraction can be acquired by using an operating system (OS) like Linux compared to bare metal programming.

1.1.1 Advantages of Ada

Sometimes systems are connected to other types of systems, in a network, forming what's is also known as cyber-physical systems [8]. An example of this is platoon driving of trucks, a technique used to reduce air resistance, which translates into lower energy consumption. A significant challenge is the security issues that arises from the openness. The trucks could be hijacked remotely, causing a disaster. These issues adds to the complexity of system, the developers needs to put an great deal of effort for ensuring not only the safety but also the security. Ada is more secure than other common languages like C and C++, for example one feature is its built-in protection against buffer overflows [14].

Embedded system development requires specific domain knowledge that needs to be combined with software development, it is therefore often control engineers and mechanical engineers who writes the software since they understand the physics of the system [15]. A great challenge is how to efficiently combine the expertise from both fields [15], work is being done and methods like contract design have been proposed [16]. Using a programming language that supports these design techniques would be beneficial. The great expansion of the embedded field have created a shortage on experienced real-time programmers, the productivity is not increasing in the same rate as the code size [17]. Managing the complexity of the software is considered to be one of the greatest technology challenges [18]. This can be addressed by using Ada with its features like contract programming and object-oriented programming (OOP).

1.1.2 Advantages of Linux

The next generation of systems have functions that are highly interactive, but at the same time needs to be isolated, like the case for autonomous systems [19], which can be provided by Linux which both isolates applications and offers fast, safe communication paths. Putting these functions on a common hardware platform when its possible will provide a performance advantage over the federated approach, specially concerning the exchange of data between functions. A CAN bus has for example a maximum bandwidth of 1Mbit/s, this is in the order of 1000 slower than DRAM speed and 100 000 times slower than cache memories in microprocessors [20]. The integrated approach enables systems to be realized where the functions are highly coupled with high demands on bandwidth and short latencies for exchanging data.

The open-source aspect is becoming an all more important factor when choosing operating systems, these OSes are now the most used for these applications, surpassing the commercial OSes [18]. The code quality for open-source projects are better than for proprietary software at all project sizes as reported by the Coverity Scan report [21].

As systems become even larger the reuse of code becomes more important, already more than 60% is reused for projects in embedded systems [22]. Linux, an open-source OS, is widely spread and for which there are a ton of open-source software readily available for anyone to use. Vision systems, A/V decoding/encoding, HTTP, web-applications, databases, security software, networking, HMI and more has already been developed. This is very beneficial for software developers that needs to implement functionality in a limited time frame.

The convergence of mainstream (PC,server) and mission-critical (embedded) market [23] put demands on the operating system to support both real-time and general-purpose applications. For those systems an operating system need to have both good real-time and general-purpose performance, but it's difficult to achieve everything at once. Linux in its standard configuration is not suitable for real-time applications, consequently solutions made by others have been developed to make it more real-time, see Section 5.2. This opens up new opportunities to develop systems with requirements on short deadlines, a lot of functionality and high throughput.

One way to run non-real-time software developed for Linux, or Windows, side-by-side with real-time tasks is to use hypervisors and virtualization techniques. By using this approach a general-purpose operating system (GPOS) and a real-

time operating system (RTOS) can run on the same hardware platform. But this means that the real-time applications needs to be programmed using the interfaces for the RTOS, a significant disadvantage compared to using a real-time modified GPOS which provides better portability [24]. Further, the performance decrease can be substantial using hypervisors, in [25] are nine times longer latencies reported, and the view of [15] is that vitalization techniques are not feasible for embedded systems because of resource constraints. With this in mind a better way to go is directly modify Linux to be more real-time capable.

1.1.3 Combined Effect

Ada is a programming language with technical features that are beneficial for development of applications with real-time, safety and high security demands. With a real-time enabled version of Linux there is the possibility of writing real-time applications in Ada that can run on the same platform as general-purpose applications, with high-speed data transfer paths between them, without risking the safe operation of critical tasks. Linux with its open source Ada compilers and presence in many domains opens up a wide range of hardware targets. The combined solution is highly portable, the design of Ada allows applications written for other OSes to be used on Linux with no or very little changes. Adding to this is the portability of Linux that allows the developer to switch hardware platform if required, due to for example performance requirements, without having to rewrite the code for the applications, whether its C or Ada.

1.2 Implementation Challenges

There are as mentioned ways to modify Linux to become more real-time capable, determining the optimal one with respect to real-time code written in Ada requires gathering and processing a lot of information. Each solution has it's advantaged and disadvantages, the relevant factors for a combined GPOS RTOS system needs to be identified. This involves acquiring detailed knowledge about both the hardware and software, how it affects performance, real-time behaviour and the effort required to use it. When the optimal candidate is identified the challenge becomes to successfully compile it, get it to run on an embedded computer. Then to develop the metrics and gather enough data to be confident in the predictability and stability of the combined solution. At the same time it needs to be user friendly, since the intended users aren't Linux experts, but instead people that are experts in other fields than software.

1.3 Contributions

The contributions of this thesis are:

- An up-to-date review of techniques for making Linux real-time and which the optimal one is from an Ada perspective, see Section 5.
- A set of factors which are important to consider when choosing a real-time Linux solution, this can be used as a checklist when evaluating a solution, see Table 2.
- A method, based on open-source tests and benchmarks, for determining what type of applications that impacts the latencies. This can be used by developers to devote resources where its needed the most, instead of optimizing code that in the end wont have a significant impact on the system, see Section 6.3.1.
- An analysis of Ada; its current state, reasons for why it is in that state and suggestions on how to increase its popularity, see Section 2.1.

1.4 Outline

Chapter 2 starts of with discussing programming languages, in particular Ada. It continues with an investigation of how modern computational hardware impacts software development, then ends with sections about real-time and operating systems. Chapter 3 describes how the research have been conducted, with details of the literature review in Chapter 4. Chapter 5 explores the different solutions for making Linux real-time. It is followed by a evaluation and verification of the optimal solution in Chapter 6. A summary of the findings and conclusions is found in Chapter 7. The appendices contains configuration details and scripts for the benchmarks. They also contain details from the literature study.

2 Background

2.1 Programming Languages

2.1.1 Ada

Ada is a programming language that was originally created by the US Department of Defense with the purpose to replace and reduce the amount of languages within the department. Now it's an ISO standard with the latest being Ada2012, typically used in the aerospace and defense sector.
By conducting a literature study, with details found in Appendix C, the conclusion is that Ada is in many ways better than other popular ones like C, C++ and Java, see Table 1.

Some of Ada's beneficial features are:

- Built-in concurrency
- Strongly typed
- Mixed language support
- Run-time checks
- Built-in methods for subsetting the language
- Supports object oriented programming
- Scalar ranges

Table 1: How suitable a selection of programming languages are for different types of systems.

	Safety-Critical	Real-time	Embedded	Desktop
Ada	High	High	Medium	Medium
C	Low	Medium	High	Low
C++	Low	Medium	Medium	High
Java	Low	Medium	Low	High

The factors used in Table 1 for evaluating the languages are defined as:

- **Safety-Critical**: The correctness of the software is of essence, faults are not allowed, since severe injury or loss of life otherwise could occur.
- **Real-time**: The language support for concurrency and predictable execution, this involves both soft and hard real-time systems.

- **Embedded**: The efficiency of the language in terms of CPU usage, memory footprint and interaction with low-level hardware.
- **Desktop**: Producing software for desktop use, usually with graphical user interface's (GUI). For example, word editors, statistical tools, games, web-browsers, communication software. Compared to the other types of systems are the demands on correctness and efficient resource usage lower, in return for faster development times.

2.1.1.1 Example Code

For complete details of programming in Ada the reader is referred to [26–30]. In this section is example code provided that highlights some of Ada's features.

One way that Ada supports contracts is by *pre* and *post* conditions, these are conditions that are checked on entry and exit of the function calls. If the conditions are not fulfilled an exception in raised. If better run-time performance is wanted, these checks can be turned off when the software has been properly tested. By having this built-into the language it provides an easy way for programmers to implement the contracts which otherwise could have been skipped because of for example laziness. Below is an example where contracts have been implemented for the Pop and Push functions of a stack implementation [26].

```
package Stacks is
    type Stack is private;
    function Is_Empty(S: Stack) return Boolean;
    function Is_Full(S: Stack) return Boolean;
    procedure Push(S: in out Stack; X: in Integer)
      with
        Pre => not Is_Full(S), -- This is the Pre-condition.
        Post => not Is_Empty(S); -- This is the Post-condition.
    procedure Pop(S: in out Stack; X: out Integer)
      with
        Pre => not Is_Empty(S),
        Post => not Is_Full(S);
    function "=" (S, T; Stack) return Boolean;
private
... --implementation of the functions.
end Stacks;
```

Ada provides the option to declare ranges for variables, if a value outside the range is assigned to the variable an exception will occur.

```
Speed : Integer range 0 .. 100;
```

Ada has built in support for task management, below is an example for implementing periodic tasks with different priorities [31]:

```
--Car code example
package Ada_Tasks is

    task Tdisplay is
        pragma Priority(12);
    end Tdisplay;

    task Tspeed is
        pragma Priority(11);
    end Tspeed;

    task Tengine is
        pragma Priority(10);
    end Tengine;

    ...

package body Ada_Tasks is
    task body Tspeed is
        Next_Time : Ada.Real_Time.Time := Clock;
        Period : constant Time_Span := Milliseconds (250);
    begin
        loop
        -- Do the job
        Next_Time := Next_Time + Period;
        delay until Next_Time;
        end loop;
    end Tspeed;

    task body Tengine is ...
    task body Tdisplay is ...
end Ada_Tasks
```

2.1.2 Ada Case Studies

Arguments about programming languages based on its features can be used to get some sense on how a language will perform. However that information won't be very reliable since there are many factors impacting the results. Case studies are necessary for determining how the languages actually perform. Two Ada studies found in the literature are summarized below, one from the industry and the other from academia. Both studies shows that programming in Ada results in less bugs compared to C, the development time and cost are in Ada's favour.

Industry study [32]: In this study a company started using Ada instead of C for a project. Before joining the project 75% of the programmers had done substantial programing in C only and 25% of them in Ada. The project was large,

complex and spanned over years. There where 70% less internal fixes and almost 90% less bugs for final customer when using Ada. Ada was not more difficult to learn than C. The final results showed that the cost of coding in Ada was about half of C, and their conclusion was that this difference would be even bigger for other organizations because their organization had good understating of the sources for faults in C and used aggressive approaches to avoid these problem. Further, their experiences was that bug rates for C++ was running even higher than C.

Academia study [33]: In a course on Real-Time embedded systems a railroad model was used for several years as a mean for teaching programming languages. For the first six years of the course there were no student group that completed the project, even though in the end more than 60% of the code was supplied to them. They used C as the programming language, a switch was then made to use Ada instead. After the first year 50% of the students completed the project, and the next years the completion rate remained over 75%! This while providing the student with less then 20% of the code. The foremost reason for this was believed to be Ada's way on handling scalar values(integers, real numbers). Mistakes in C that took a full day to find was immediately reported by the Ada run time systems, these mistakes are so common that it makes a big difference for the end result, many of the problems with scalars that arises when using C was said to be non-existent for Ada.

2.1.3 Factors for Choosing a Language

There is a question whether programming should be done manually at all, or if the code should be automatically generated from models. The model driven approach benefits from using intuitive graphical notations that are more expressive and provides higher level of abstraction [15]. There is a trend in moving from manual programming to model-driven development, however a lack of tool support makes it impossible to cover the complete life cycle using model-based methods [15].

Automatic code generation has been a several decade long dream for software engineers and managers, but there are still significant challenges like the execution speed and memory usage [17]. Then there are the problems of transitioning to and from design phases that was pointed out by Glass in the mid 90's, the opinion was that it would make large scale code generation extremely unlikely [34]. These challenges are still valid since there has been no breakthrough in automatic code generation according to [17]. Model driven engineering and automatic code generation could very well be the future, but for now the systems

can't deal without manual programming.

When selecting a programming language one needs to consider several factors. There are technical ones such as efficiency, concurrency support,footprint, maintainability, syntax, portability, real-time support, safety and hardware access [35, 36]. These technical factors are not the only ones, and sometimes not the most important either. The non-technical factors like the availability of programmers, compilers, support, documentation, training courses and online communities plays an important role: "*The reason for choosing a particular language may ultimately be based on factors having nothing to do with the technical merits of the language itself*" [37].

A lack in either type of factor, technical or not, can result in not choosing a particular language. If the language does not have the technical features to solve the problem it does not matter if it's available on every single platform with an abundance of skilled programmers. At the same time it does not matter if a language is technically superior when there are no programmers or compilers to be found.

Selecting a suitable programming language is an important step in the software development [14, 38, 39]. On the other hand McDermid [40] argues that programming language has little bearing on failure rates of safety-critical software. He *knows* of a system written assembler that has had over 20,000,000 flying hours without hazardous failures. This is anecdotal and perhaps not the best evidence, but lets assume that the hazardous failures does depend on the programming language, does that mean that the language is unimportant? No, it doesn't, since safety-critical systems, such as those in aviation, needs to be certified against some standard [41, 42]. This process should make sure that the software has a reliability of some degree, for example it must be proved for the highest level safety that there will be a maximum of one failure in 10^9 operation hours [43, 44]. Another important aspect is the evolution of complexity and scale of projects. The code size today is many times larger than it was a decade ago, see Figure 1.

Nevertheless, in principle any language could be used, but choosing one that is designed for reliability, safety, and real-time will reduce the effort and cost [14, 45–47], especially during the operation phase: *"For complex code that must be maintained over many years, maintenance costs typically run 70% to 90% of the software life cycle costs."* [37].

What factors are important for choosing a language? The UBM Market Study [18] shows that the most important software tools are the Debugger, Compiler

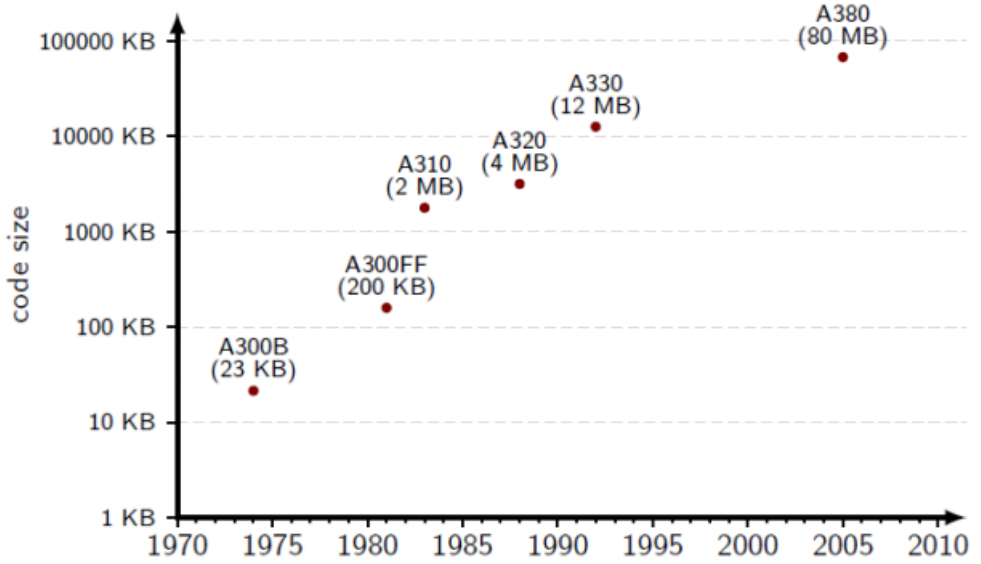

Figure 1: The code size have grown substantially during the years for Airbus aircrafts [48]

and IDE, in descending order. Source code analysis tools, testing tools that is important for safety-critical development are however in the bottom of the list. This is supported by the VDC survey [49] which shows similar results. These studies uses data form many domains of the embedded market, like automotive, aerospace, medical, industrial automation and consumer electronics.

Meyerovich et al [50] analyzes open source projects for the purpose of finding factors for language adoption. The projects can be of any type; embedded, desktop, web etc. The results, see Figure 2, clearly shows that the external(non-technical) factors of the language are the most important ones for adopting a language. The top factor is the availability of open source libraries, however for large companies (+100 employees) the possibility to build onto existing code is the most important factor. Intrinsic factors like simplicity, language features and safety/correctness ranks low. This shows how difficult it is for small languages to become more widely used. A language needs to be popular in order to be chosen, but no language is popular from the beginning. The technical features must really be much better than for existing languages to outweigh the lack in other factors like familiarity.

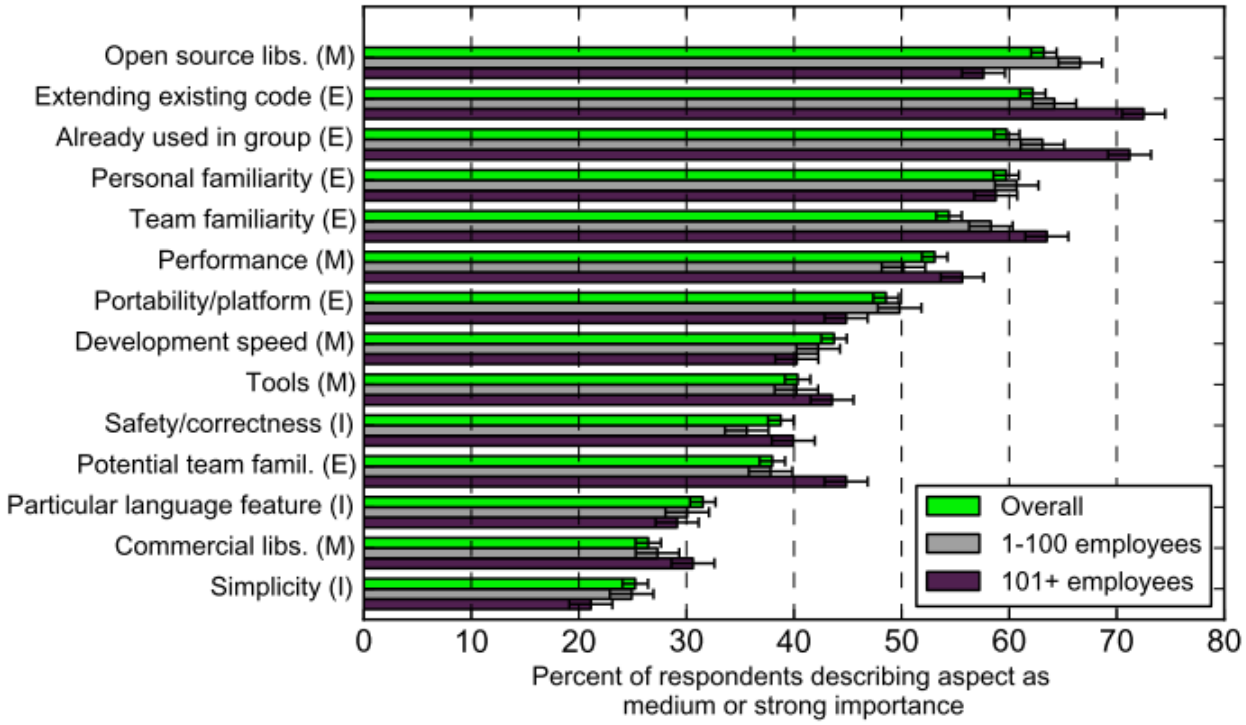

Figure 2: How companies of various sizes values different factors when choosing a programming language. E = Extrinsic factor, I = Intrinsic, M= Mixed. Figure from [50]

.

2.1.4 Popular Languages: Surveys

Ada is often mentioned in the literature as being a popular language for safety-critical systems, see Appendix C, but how popular is it according to large scale studies?

IEEE Spectrum [51] tracks language usage for different domains like embedded systems and web applications. It is weighing data from different sources like Google searches, Github projects, Stack Overflow, Reddit etc. In the embedded domain C is the top language, followed by C++ and then Assembly. Ada is only ranked at 10 of 13, when considering the Trend or the Job ranking it comes in at 7/13. When all domains is considered Java tops the list, followed by C and C++. Ada comes in at 35/49.

The TIOBE Index [52], which considers all type of systems, shows how Ada has steadily declined in popularity since 1995, from a Top 5 language to top 30. C

and Java is battling for the first place, with about 16% each of the market. C++ is close to 8%, Ada comes in at 0.7%.

The embedded market studies shows the same situation, Ada is far from a dominating language. According to the UBM Market Study 2014 [18], Ada was used by less than 1% of the developers, with the main language being C. The VDC report [49] from 2011 shows however that Ada+SPARK usage was roughly 5%, this may be due to the higher amount of respondents from Military/Aerospace in which Ada has its strongest position: A survey of languages used in Department of Defense showed that Ada was the most used language [53] in the mid nineties.

Chen et al [54] analysis 17 different programming languages, one interesting result is that their survey from 2003 shows that the top five popular languages was C++, Java, Smalltalk, Ada and FORTRAN when considering the grass root support, notice that C is not on that list. According to their result C went from being the most popular language in 1998 down to 6th popular in 2003. This is contradicting the results from the TIOBE Index which shows C as the top language in both 2000 and 2005. Depending on the respondents, the domain considered, if companies, hobbyists or universities are asked the results will be different, sometimes significantly like between the TIOBE and Chen study. Based on this it's not possible to say which language is the most popular in general terms, however it is possible to draw the conclusions that Ada is a niche language.

2.1.5 Why is Ada not More Used?

The studies above gives some indication on why Ada is not very popular. The lack of popularity is a strong reason in it self for not choosing the language. Popularity breeds popularity. What other reasons can be found?

If we start at the point where many programmers will begin their training, at the academic institutions. The universities are providing the industry with educated personnel, obviously the choice made by them will influence the availability of programmers. What languages do they teach? Why do they teach those languages?

The choice of programming language to teach has been debated since it was first introduced in the university curricula [55]. The main question is whether to choose a language because it's pragmatic or pedagogical. The pragmatic approach plays into the industry demands and focuses on student employment, the pedagogical approach focuses on the suitable to teach programming concepts and problem solving, minimizing the effort spent by the student to understand

complex programming languages [55, 56].

> “A language that requires significant notational overhead to solve even trivial problems forces the language rather than the techniques of problem-solving to become the object of study
>
> John M. Zelle [57].”

Institutions like US Air Force Academy (USAFA) recognized that language like Java and C# was much more widely accepted in industry than Ada. However they saw that it was their obligation to use the best tool available, which they considered to be Ada, to teach new computer scientist. They however mentioned that their academy was not as affected by industry like civilian institutions, thus having more freedom to select an appropriate language [58]. This seems however to have changed since Ada is no longer taught as the introductory language at USAFA, but it's still being used by the US military academy (USMA) and as well by the faculty for robotic research [59]. It was chosen as introductory language at MIT, but here one important argument for Ada was that it's actually used in the industry, namely the aerospace sector [60]. On the other hand, in [3] it is said that Airbus considers the education by schools in aeronautics curriculums is to low level, not sufficient to handle the complex systems. Instead they say training is done in the company by mixing experienced and new personnel in the same teams.

In [61] trends and reasons of teaching programming languages are discussed by looking at 4 studies from 2001,2003, 2010 and 2013. These studies builds on data from Australian and New Zealand, showing the most popular languages and the main reasons for choosing to teach them. In 2001 the top reason for choosing a language was its marketability, its use in industry, second to that came the pedagogical aspects.

By 2010 there had been a shift, the marketability factor declined from 56.1% to 48.8% and the pedagogical increased from 33.3% to 53.5%. From the first three surveys it is not possible to determine how important the respondents thought a specific aspect of the language was since they was not asked to rate them. The 2013 study did however ask respondents to rank how important a specific reason was. This showed that it was the pedagogical and platform independence that was the most important, the industry aspects was not as important, see Figure 3.

How has this been reflected on the choice of language? In 2013 Java and Python was clearly standing alone at the top, both at 27.3% when considering number of courses. When consider amount of students that are being taught the language

Python emerged as the winner with 33.7% of the share compared to Java with 26.9%. What makes the figures more astonishing is that in 2003 Java was the clear top language with more than 40% of the students/courses, Python was a new language and scored 0%. The new trends in choosing a teaching language have benefited Python and been a disadvantage to Java.

What about Ada? Well the studies shows that only 2.3% of the courses used Ada, which makes it a bottom language. The trends is however positive, and Ada has the highest rank now since 2001. The studies does not go further back than 2001 but another paper from 2000 [62] shows how the amount of colleges teaching Ada peaked in 1997 and then remained at that level until 2000. This while the tool/vendor trend was quickly declining. They thought this was due to academia is slow to change. This could be the reason why Ada in the 2001 report was still in use, but the disappeared from the curriculum in the 2003 and 2010 report.

What is the reason then for Ada being introduced as an teaching language again? Since the popularity in the industry aspect is losing its importance for the choice of language, Ada can be considered for it pedagogical benefits. Another important factor is the availability of free compilers and developing environment [63], this played a role when Ada was chosen as a language in a university in Germany [64], and is as well one of the factors that respondents thought was important when the choosing Python as a language in the [61] study.

The results from [61] is not applicable across the world since the data comes only from Australia and New Zeeland, but [65] shows that the trend of moving from Java towards Python in the introductory language is present in the USA as well, the same goes for the use of Ada which is at 1.7%. A survey [66] of 39 colleges in the mid-west USA showed that approximately 9 programming languages were used in introductory courses, some of them were C, C++, Alice, VB, Python, Java, C# and Ada, but they did not provide to what extent different languages were used. More information about language choice can be found in literature surveys like [56, 67, 68]. However there is a lack of systematic large scale studies [68, 69], but until there such studies available the previously ones discussed gives at least a view of the situation in USA, Australia, New Zeeland and to a small degree Europe. The indication is that universities are pulling them self away from being puppets of the industry, which should be comforting for some:

> “ The greatest danger to our university system is the lemming-like rush to do the same thing, to be one with the crowd, to be part of the current fashion industry of computing.

Johnson [70] ”

Indeed, the side effects of trying to teach a language that is used in industry but that is not best pedagogical have been pointed out in [71]. The presented view discourage the use of Java because it teaches students how to pick bits and pieces together from the vast amount of libraries and frameworks. They say it results in students being able to put programs together but they become less skilled at actually programming, the concept of run-time costs are lost. From private conversations that they reference to it is revealed that many big companies like Intel, Microsoft, AT&T, IBM, Lockheed Martin and more are unhappy with the approach to teach Java as an introductory language.

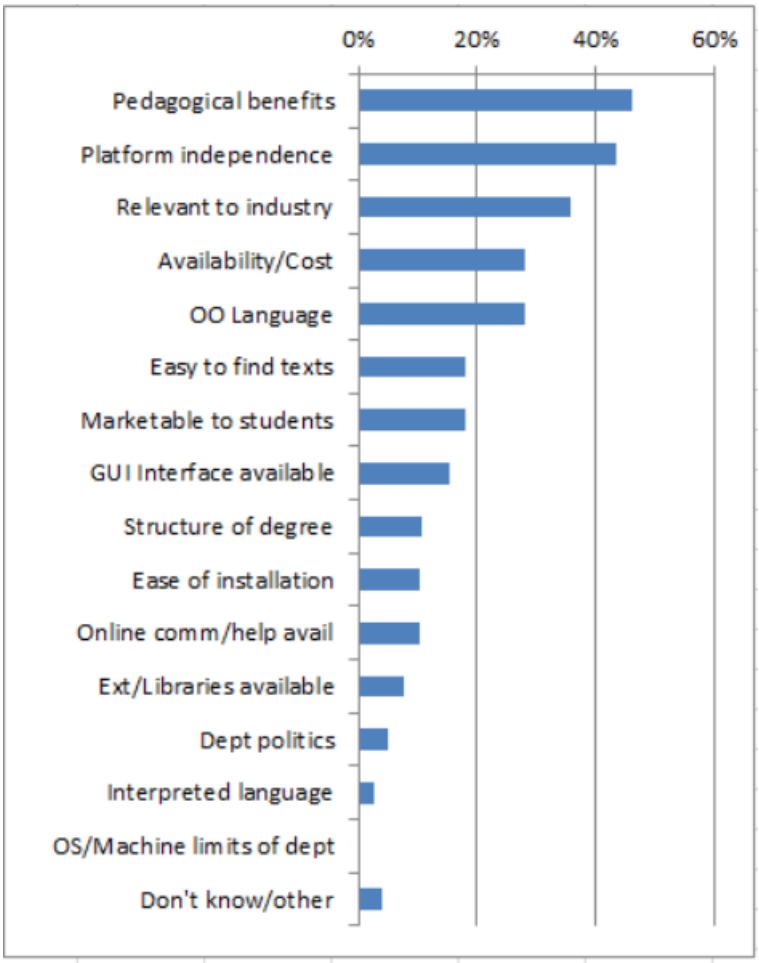

Figure 3: Important aspects that are considered when choosing a programming language to teach. Figure from [61].

2.1.6 Summary

There is a trend of software increasing in both code size and complexity, the object-oriented language like Ada, Java, C++ suits well with the demands of productivity, maintenance and code reuse. Ada takes one step further in the support of building systems using the correctness by construction approach, which is becoming more important [19]. But for some high performance applications it will be necessary to use OOP languages in combination with C or Assembly [17]. Ada has a good mixed language support which could accommodate such demands.

It is clear that Ada is a niche language used for safety-critical, large scale and real-time applications in aerospace and military. Its position as a mainstream language is weak, often non-existent in some domains. In education its position is weak as well, however the trend of caring less about industry demands and more about pedagogical aspects has re-introduced Ada as an option. The organizations behind Ada has an opportunity here, if they can devoted resources for pushing Ada into the curriculum the language might gain the necessary momentum to make it a real contender in other domains than aerospace and military.

The availability of open-source, free compilers for Linux has aided in a mini-comeback. Ada has always been a technical superior language in many cases, the latest addition of support for contract programming builds on its strengths even further. However as can be seen by the studies above the external factors of a language is many times the most important ones. People, companies and institutions value very strongly the availability of compilers, software libraries, and familiarity of the language. Ada has in the past scored low when it comes to compilers, platform support, libraries, tools and familiarity. But things are turning around for some of these aspects.

In any case it is unlikely that C will continue to be a dominating language due to is lower abstraction levels compared to other languages. As [12] describes the language trend: In the early days Assembler took the step up from machine code, later in the 70's higher abstraction was provided by languages like C and now we have object-oriented languages to further make programs easier to write and understand, increasing productivity.

The desktop domain has already shifted to object-oriented languages as can be seen by market studies above, the embedded domain that is more resource-constrained is still dominated by C but probably not for long considering the trends. The time-to-market and rising complexity are main factors for transitioning to other higher level languages, especially since the software development is

the dominating cost factor for embedded systems [22].

2.2 Computational Hardware - Challenges

The ways of increasing the performance of hardware the last decade is different to what it has been in the past which complicates things on the software side. Software now needs to be parallel to take advantage of the performance increasing features, something that turns out to be very difficult. The development of real-time systems have also been affected, in a negative way. The hardware is becoming less timely predictable, which makes the traditional techniques for ensuring results arriving at specific deadlines less effective. This chapter provides details of how the hardware have been changed, the trade offs that are being made and how it affects real-time systems.

In the past, for almost 20 years, the performance of microprocessors was increased by approximately 50% every year [72], see Figure 7. Software engineers could sit back and rely on hardware engineers to provide increased performance of their programs, but around 2005 this changed: "*The La-Z-Boy programmer era of relying on hardware designers to make their programs go faster without lifting a finger is officially over. If programmers want their programs to go faster with each generation, they must make their programs more parallel*" [72]. The traditional way of improving performance by increasing clock frequencies and instruction level parallelism (ILP) has more or less come to an end, the methods for the processors to do parallel work without requiring changes in the software design has been exhausted, and the increased power consumption put an end to increasing clock frequencies [73].

> “ The transition towards chip multiprocessors has created a disruptive change: For the first time in many years, substantial performance gains for applications can no longer be achieved without modifying the underlying source code. Future applications must be radically different - they must be parallel.
>
> Huard et al. [74] ”

Further the memory speeds has not kept up with processors speeds, see Figure 4, for example the memory bandwidth of DRAM is only 6% of what the core i7 Intel processor can utilize [72], unaddressed this would have been a serious bottleneck of modern processors. To mitigate this smaller faster memories, cache, are built into the processor. Depending on the application the effect of cache will differ, an typical case is shown in an example by [72], with no cache the system would be 40 times slower than the one with cache. This shows the waste of resources that

occur when real-time developers in the old-fashioned way disable cache [75] to make worst-case execution time (WCET) calculations possible.

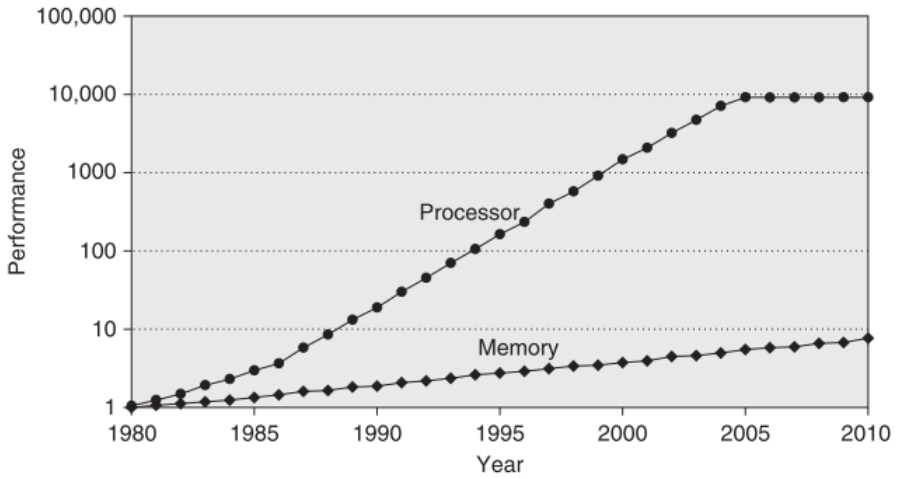

Figure 4: The DRAM performance, in terms of latency access, have not kept up with processor performance, in terms of memory requests. The processor performance levels out at 2005 since this graph shows performance on a per-core basis. Figure from [72].

As can be seen in Figure 5, there is a trade off between speed and capacity, the memory closest to the processor is the fastest but at the same time the smallest. The reason for slow DRAM is however more due to a practical matter than a technical one; storage capacity and low cost has been prioritized in exchange for speed, since this have been possible due to cache memories [73, 76]. A problem with cache is the storage capacity, for it to be useful the hardware or software have to make predictions on what data that will be used next, so it is available when requested. When data is needed by the CPU the first place to look for it is in the cache, if it's not there the CPU has to look in DRAM. This results in a substantial performance penalty, since the latencies can be 100 times longer for DRAM compared to cache as shown in Figure 5 .

The hit ratio for data requests from cache must be close to 1 to achieve any significant speed up [77]. In [72] it is shown how for example a computer with no cache misses is 1.75 times faster than one with 2% misses. By using different data replacement strategies and clever programming it's possible to make a significant difference on the hit ratio [72]. However for some application like Big Data, the memory demand is substantial and often the data cannot be found

in cache, this puts demands on new memory architectures that have both large storage capacity and performance [78].

Level	1	2	3	4
Name	Registers	Cache	Main memory	Disk storage
Typical size	<1 KB	32 KB–8 MB	<512 GB	>1 TB
Implementation technology	Custom memory with multiple ports, CMOS	On-chip CMOS SRAM	CMOS DRAM	Magnetic disk
Access time (ns)	0.15–0.30	0.5–15	30–200	5,000,000
Bandwidth (MB/sec)	100,000–1,000,000	10,000–40,000	5000–20,000	50–500
Managed by	Compiler	Hardware	Operating system	Operating system/ operator
Backed by	Cache	Main memory	Disk	Other disks and DVD

Figure 5: The latencies and bandwidths for memory increases with the distance from the processors. There is a trade off between speed and size resulting in several layers to achieve optimal performance in terms of performance, cost and energy consumption. Figure from [72].

Since the single core performance stalled, see Figure 4, the concept of multi-core processors were introduced (Figure 6), this has enabled the continuation of performance increases, though at a slower rate, see Figure 7. However, then end of single-thread performance and the way to increase performance by parallel means has perhaps created the greatest software challenge in history of computing according to [76].

The ways to improve performance for microprocessors have focused in increasing the average performance, while this works well for desktop and server applications it is not the most important factor for most real-time systems where predictability and worst case performance is typically what matters [79].

Before the introduction of multi-core processors the real-time systems developers already faced significant challenges; Pipelines, cache's and branch predictions are good means of improving average performance, but the CPU becomes less deterministic [13, 80], see Figure 8. 10 years of research between 1995 and 2005 did not improve the correctness of WCET, it actually became worse [81], which is pointed out as actually being huge progress since the architectures changes are just making the WCETs more and more difficult to compute.

The traditional response time analysis methods are often too pessimistic to be

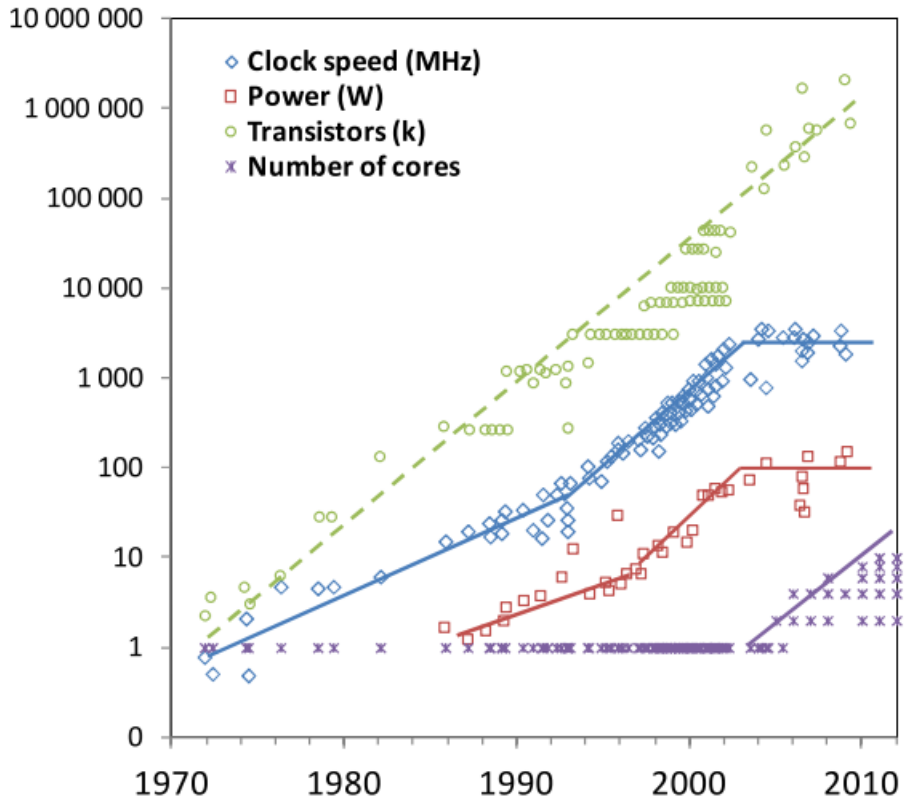

Figure 6: The amount of transistors has increased over the years improving the processor performance. A very important aspect is that since 2005 the extra transistors has been used for implementing multiple cores since the single-core performance can't be increased in a significant way by adding more transistors. The clock frequencies can't be increased due to the power consumption. Figure from [74].

useful [82]. The introduction of multi-core processors just makes these calculation even harder: "*To map real-time tasks onto the processor cores for system-level resource management and integration, a large number of scheduling techniques has been developed in the area of multiprocessor scheduling. However, the known techniques all rely on safe WCET bounds of tasks. Without proper spatial and temporal isolation, it seems impossible to achieve such bounds. To the best of our knowledge, there is no work on bridging WCET analysis and multiprocessor scheduling*" [83].

According to anecdotal information referenced by [8] the real-time performance of PC's haven't improved since the mid 1980's, which is however noted to also be because of overhead when using for example operating systems. There are of course ways of making the architectures more predictable, [84] presents an approach to design time-predictable chip-multiprocessors and [13, 83] concludes

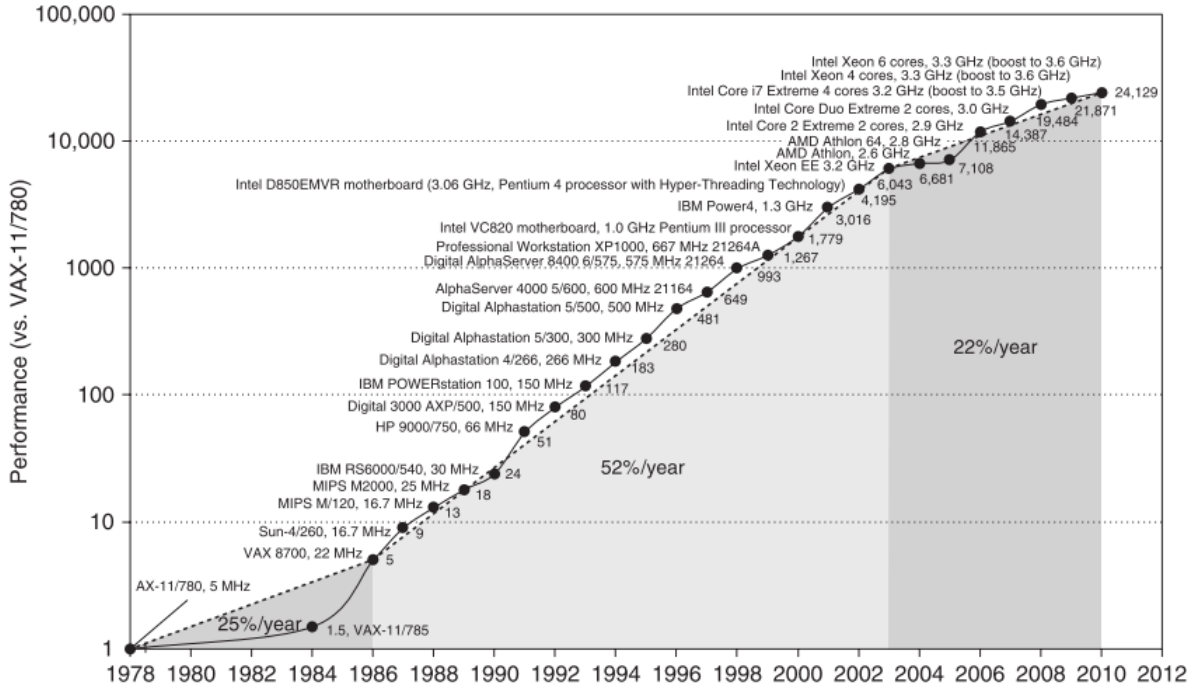

Figure 7: For many years the processors performance was increased by 52% per year, but this has now slowed down, and more importantly the increases now are mainly because of the implementation of more cores which causes problems at the software level related to parallelism. Figure from [72].

that more predictable architectures are needed in order for being able to perform useful timing analysis. The question is whether microprocessors vendors will listen to this and start developing time-predictable architectures, it is not pointing in that direction since benchmarking is focusing on average performance, the timing predictability is not considered [8]. Also [23] notes that the relative low-volume of processors with these requirements would be extremely costly because of Non-Recurring Engineering (NRE) costs.

On the other hand, [11] says the typical worst-case design approach is no longer acceptable because of dynamic environments, that approach is too resource wasteful, instead adaptive techniques needs to be used, where the system can detect and adapt to changing environments. Indeed, visions systems for example have very varying load depending on the objects in the field of view [85]. Another way demonstrated by [86] is to use health-monitoring systems that can complement systems building on uncertain WCETs, as they mentioned that tight WCET calculations are difficult, sometimes providing overestimates of 100%, and for

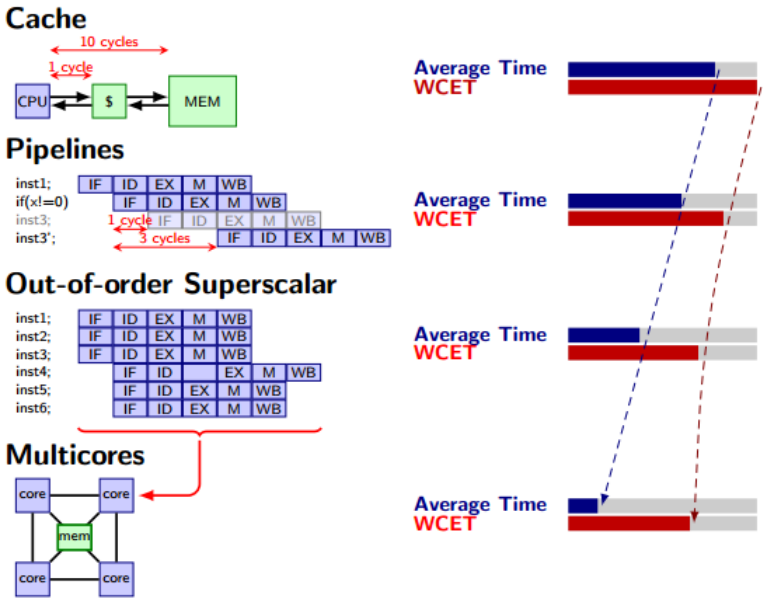

Figure 8: The average execution time (ACET) have increased significantly due to improvements in the architecture, however the important worst cased execution time (WCET) property for real-time systems have not improved as much, resulting in an larger gap between ACET and WCET. From [23]

some processors being impossible. By monitoring the tasks and detecting errors a safe routine can intervene in case it's needed. These ways of working around the problem of unpredictable hardware layers does however rise the complexity that is so important to reduce [8], but in the mean time while microprocessor designers does not provide predictable hardware the only way forward is to use what is available.

2.3 Real-Time

2.3.1 What is Real-Time?

In the introduction chapters of research papers and books that involves real-time systems there is often an definition of real-time systems in the style of: "*The correctness of a real-time system depends not only on the logical result of the computation, but also on the time at which the results are produced*" [5, 24, 29, 87–92]. This is unfortunate because that definition is not very useful in the sense it doesn't make a clear distinction between real-time systems and other systems. For all systems there will be a limit for how long time that can elapse before a result is received, but it might not be explicitly expressed. A result that is

produced after a million years won't be useful, real-time or not. When browsing the web there might not be any explicit timing constraints for when the result needs to be produced, one or a few seconds could be okay, but if it takes one day it would in most cases render the result useless. But how many will classify web browsing as a real-time system? A better definition, is the one presented by [93]:

> “ A real-time system is one in which the temporal aspects of its behaviour are part of their specification. The correctness of the system depends not only on the logical results of the computation, but also on the time at which the results are produced.
>
> Bernat, Guillem [93] ”

The first sentence makes all the difference, if the timing of the result is important enough to be part of the specification it is a real-time system. Timing properties will in many cases not be part of the specification for web-browsing, but in some cases, like stock-related information, the timing is very important and it is not unlikely that the developers of those systems put an effort into making sure that the result is delivered in a specific time frame. If that is the case it is a real-time system. This definition, will probably not be spot on for 100% of the cases, but it's the one that will be used here. This is also in the lines with the German industry standard DIN 44300 (1985) that [94] references to, *"[A real-time computer system is one] in which the programs for the processing of data arriving from the outside are permanently ready, so that their results will be available within predetermined periods of time"* [emphasis added].

Real-time systems can be classified as being hard, firm, soft, weakly hard. Often only the hard and soft notations are being used, at least in the papers reviewed by this thesis. The definitions for real-time systems used here are:

Hard
In hard-real time systems a missed deadline could lead to catastrophe, hence no deadlines are allowed to be missed. [5, 17, 30, 91, 95, 96]

Firm
The result received after a missed dealine has no value, but some amount of missed deadlines are tolerated [17, 30, 95, 97]

Soft
A missed deadline is degrading the performance, but the result still has value. [17, 91, 95, 96]

Weakly hard
Weakly hard systems can be seen as a more elaborate version of firm real-time systems. A firm real-time system is vague in the sense that it does not specify how many dealines that can be missed in a sequence. Missing one deadline every 100 is different compared to missing 100 deadlines in a row. Weakly hard systems specified how many deadlines that can missed during a specific time window [93, 98].

2.3.2 Real-Time Implementation Challenges

The Ada+Linux solution running on a modern hardware platform will have difficulties to support traditional ways of developing real-time systems. Hard real-times are very difficult to guarantee, probably infeasible.

An essential part of many real-time systems is the scheduling of tasks, there are different algorithms for this and according to Buttazzo [97] the Rate Monotonic (RM) and Earliest Deadline First (EDF) are the most common ones, of which EDF is supported by Ada. There are misconceptions on which is the most optimal one and how they handle overload conditions, Butazzo performs simulation experiments that shows how the assumption of RM having better jitter control and more predictability for overload conditions doesn't hold in general.

To be able to schedule tasks for RM and EDF the time it takes to execute the task, including context switches etc needs to be known. As mentioned before, this becomes problematic for modern complex microprocessors where timing aspects are extremely difficult to have full knowledge of. One problem is the so called timing anomalies, one might think that computing the worst case scenario for parts of the system and then adding them up will give the global worst case scenario, but this is not the case. As explained by [99], for advanced architectures a local worst case like the cache miss can actually result in lower global worst case execution times compared to the much faster cache hit. To be able to find the WCET for these architectures they say that all execution paths needs to be analyzed, but for complex systems full-path coverage is infeasible [81, 100].

If the application does not require state-of-the-art performance simpler architectures can be used, like the older ARM7, for which only the worst case scenarios of the sub systems need to be considered, hence full-path coverage is not needed [99]. A less safe method is just to simply run the code and measure the execution time, the observed WCET will unlikely be the real WCET, nevertheless this method is the most common in industry even though it's unsuitable for hard real-time systems [81].

The solution to the WCET problems can be the Measurement Based Probabilistic Timing Analysis (MBPTA) technique that uses extreme value theory to give an estimate of the timing properties to any desired degree of probability. The execution times for a program is measured, and with enough samples statistical methods can be used to give the probability that a certain execution time will occur . However, this requires that the observed execution times can be modelled as independent identically distributed random variables [100], unfortunately no commercial modern microprocessors fulfills the randomisation requirements for this method [101].

Further, for dynamic systems deadlines cannot be guaranteed in advance [102]. For example, the CPU utilization for a self-driving vehicle depends on the environment; heavy traffic, multiple road alternatives, surrounding objects etc. Conventional real-time approaches fails to incorporate the dynamic nature, therefor other approaches like data-analysis and prediction of the workload is an alternative that is presented by [103]. With highly variable systems like that it would be inefficient, and perhaps impossible, to use a CPU that can deal with the worst case scenario for all tasks at the same time. This can be solved by assigning tasks different critically, when the high-critical tasks needs the a lot of the computing resources the other tasks have to wait to do their part until the workload reduces.

2.4 Operating Systems

The purpose of an Operating System is to manage the sharing of resources on a computer, it provides a set of APIs for the users to develop applications running on the OS [104]. It also provides isolation between applications, for the most of the time, only letting an application have access to its own memory space if it runs under what sometimes is called 'user space'. Some applications like drivers runs in the core of the operating system, 'kernel space', these applications has access to all the memory, making it extra important to ensure the correctness of them since a bug can crash the whole system.

For real-time systems it often desirable to run the application on top of a real-time operating system (RTOS), this becomes an necessity for large projects, for example the Mars rover had 120 tasks running in parallel on top of an RTOS [105]. However, for soft real-time applications where the throughput is more important a GPOS can sometimes be more suitable than a RTOS.

The API richness can be very different between OSes, in a paper [106] evaluating several RTOSes the amount of systems calls range from 18 to 93. The basic

services that a RTOS should provide is generally mention as being methods for task management, time management, interrupt handling, synchronization/inter-process communication and memory management [107–110].

The differences between a RTOS and a general purpose operating system (GPOS), like Linux and Windows, are typically that the RTOS has at least one scheduling method that can guarantee timing properties of tasks in a predictable way, it has predictable synchronization between tasks [104], generally the key word is determinism. The GPOS is optimized for maximum throughput, sacrificing short latencies, while the RTOS is optimized for predictability, placing an upper bound on latencies, which on the other hand can degrade throughput performance [111].

> “The goal of a Real-Time Operating System is to create a predictable and deterministic environment. The primary purpose is not to increase the speed of the system, or lower the latency between an action and response, although both of these increase the quality of a Real-Time Operating System. The primary purpose is to eliminate “surprises".
>
> Rostedt and Hart [112] ”

Other aspects that needs to be taken into consideration are the support of development tools, safety certification and error handling of system calls [113]. The middleware in the RTOS is also important, for example the file system, communication stack and portability [3]. To achieve portability it is beneficial if the operating system support standard interfaces, for example any of the four main ones POSIX, OSEK, APEX and ITRON [11]. For some applications the energy consumption can be of interest as well, one observation shows that the energy consumption of complex RTOSes are less sensitive to the workload compared to simpler ones, complex RTOSes have a higher initial higher energy consumptions, but with the benefit of being more predictable [114].

3 Research Design

3.1 Research Goals

The research goals are to:

- Provide deeper understanding about programming languages, why does people chose one before another, what is the state of Ada, its strengths/weaknesses, what is its future?
- Find out how people choose operating systems in order for determining the expected reception of the Ada+Linux+SBC (Single Board Computer) combination; if it shows superior performance will that be enough for people to transition from other solutions? What can the organizations/teams behind Ada and Linux do in order for making their work reach the masses to a larger extent?

3.2 Research question

Based on the challenges mentioned in the introduction, developing systems with highly-coupled mixed type functions on an embedded hardware platform, the research question becomes:

> **Research Question:** *What is the optimal way to modify Linux for enabling real-time execution of Ada code?*

3.3 Methodology

In order to answer the research question the task has been broken down into two main parts; a literature study to find the optimal solution and a practical part for verification and evaluation.

The literature study gives answer to what the optimal solution is by:

- Determining factors that are important when choosing an operating system, see Table 2.
- Giving answers to how the real-time modifications of Linux evaluates against these factors, see Chapter 5.
- Providing information about Ada support for those solution, see Chapter 5.

Further, it provides the contents of Chapter 1 and 2. The details of how the literature study was conducted is described in Chapter 4.

When the optimal solution has been found it is verified and evaluated in Section 6 by the following methods:

- Implementing the modifications, installing it on a BeagleBone Black.
- Verifying the Ada support by testing the compiler using the GCC Ada testsuite.
- Executing benchmarks for evaluating real-time and throughput performance, see Table 6.

Table 2: Factors for that are taken into consideration when determining the optimal solution.

Factor	Description
Ada compatibility	Is Ada supported? No solution can be considered optimal without Ada support
Performance	What are the latencies and throughput?
Portability	Can it be used on multiple hardware platforms?
Predictability	Are the functional and temporal results predictable?
Ease of use	How much effort is required from the user to take advantage of the real-time properties?
Organization	Is there an active team or stable organization developing behind the solution?
History of use	Have it been successfully used before?
Installation	How much effort is there to install it, is it difficult, are there many things that can go wrong in the process?
Community	Is there an active community where support can be found?
POSIX compliance	POSIX is a IEEE standard OS interface that many OSes uses, allowing for code to be portable.
Documentation	Is there detailed and up-to-date and documentation on how implement and use the solution?

3.3.1 Reliability

Will same results be produced if the research is repeated?

Literature study
The process of finding relevant real-time Linux solutions in this thesis is based on citation count and date of publish. These factors are dynamic, they will change with time. The databases might improve in linking research papers between each other, providing more accurate citation lists. The top 15 citing articles of a certain paper today will most likely be different ten years from now. With this in mind, the results are reproducible when considering the time that this thesis is written.

Verification and Evaluation
All the relevant software and their versions have been documented to improve the reliability. Different software versions can show completely different results, but as long as the setup is the same as described in this thesis the results should be reproducible. There is however one exception to this, the measured worst case latencies. These measurements can be considered as a function of probability. The longer the measurements occur the higher probability for the worst case to be reported. For these types of systems where theoretical analysis is infeasible it can't be guaranteed that the same latencies will be measured. The average case latencies should however not deviate significantly to the ones presented here.

3.3.2 Validity

Are the results an accurate description of reality?

When trying to find the optimal solution based on the literature study there are two main threats to validity.

- Enough relevant data has not been found.
- The data that has been found is inaccurate.

The first point can be seen as a coverage problem, has all solutions been found and considered? This is handled by the way the literature study has been conducted as described in Chapter 4. The second point is a bit of a concern when using data from computer science. There is a lack of empirical evidence in the peer reviewed literature, statements are often anecdotal and self-reported claims of efficiency are common, see Appendix D that touches this subject. The optimal solution is based on a set of factors that was derived of the information found in the literature, if those factors are not as important as the authors claim the results in this thesis will be less valid. This is mitigated by a correctness-by-numbers approach, if a view is found in several articles it is deemed to be valid. To further minimize the threat, market studies have been included which provide statistical

data, together with the peer-reviewed literature this approach is deemed to be sufficient.

4 Literature Study

In order to help answering the research question and meet the research goals a literature study has been conducted. It can be divided into two parts, one part focuses on the real-time aspect of Linux and the other on programming languages. For both parts Mendeley [115] has been used to manage the references. It was chosen because it has a built in PDF reader, support for annotations, tagging and full text searches. Some papers contain information that spans several subjects, for example one source might discuss both operating systems and programming language, using folders would therefore not be a good approach for organization. Instead tags where used to categorize sources since a source can easily be assigned with multiple tags. Later on the tags can be used as a filter, showing only sources with a specific tag. This was necessary because with hundreds of sources it becomes difficult to keep track of which one to use and where.

The tagging was done using a tree-like structure, by first using the keyword CODE and then adding sublevels separated by a underscore "_", for example CODE_RTOS_LINUX_RTAI and CODE_RTOS_LINUX_XENOMAI. Table 3 shows the main tags and the number of sources under each, note that this includes all sources that have been added to the database, not only the ones that in the end where used in this thesis.

Table 3: Tagging were used to organize papers, these are the main levels in the tree structure, a source can have multiple tags The table shows the number of sources for each tag.

Code	Amount
CODE_BACKGROUND_	67
CODE_EDUCATION_	19
CODE_HARDWARE_	13
CODE_LANGUAGES_	75
CODE_MARKETSTUDY_	7
CODE_REALTIME_	35
CODE_RTOS_	45
CODE_TESTING_	19

4.1 Real-time Linux Modifications

The main goal of this part of the study is to find the metrics and candidate solutions for determining the optimal real-time modification for Linux. The tasks

involved with this are the following:

- Step 1. Do a background study to determine what factors that should be used for evaluation of the solutions.
- Step 2. Search the literature for solutions and make a list of possible candidates.
- Step 3. Find basic information of all the solutions, exclude the ones that fails in any factor.
- Step 4. Find detailed information of the remaining solutions.

The results from Step 1 are used in Chapter 1 and 2, the results from steps 2-4 are used in Chapters 5 and 6.

4.1.1 Step 1. Background

In order to determine the optimal solution there needs to be some factors to evaluate the solutions against. A complete system consists of the hardware platform, operating system and programming language, which all affects the end result of the system. The goal of this part is to gain knowledge about real-time, safety-critical systems, general-purpose systems, operating system and hardware platforms. This knowledge is then combined with the perspective of this author to determine what factors to use when evaluating the different solutions. The sources found in this study are used in Chapters 1 and 2, which contributed to deriving the factors in Table 2.

The approach for this part has been to find enough sources for a certain topic, which is judged by the quality and the information they provide. If the views for a topic are diverse with an ongoing debate it calls for more sources compared to a topic where there is a consensus. Since the goal for this part is not to statistically present how many papers are supporting view X the method to search for information was highly dynamic, iterative and adaptive. Multiple search terms, constructed by the main keywords: *real-time, safety-critical, embedded, testing, fault tolerance, cyber-physical, integrated federated approach, challenges, rtos, rtos selection, microprocessor future, market study, caches, pipelines and certification.* A strong focus on examining references used by papers and citations, full text searches when needed, making judgement call for each paper, either stopping or keep going depending on the quality and amount of information needed. Sources for information have been found by using Scopus and Google Scholar.

4.1.2 Step 2. Finding solutions

The goal of this part is to find solutions that potentially could be the optimal one. No evaluation is done at this step, the purpose is only to provide a list which is

subject to a more focused study in a later step. However, if a solution does not have a chance to be considered optimal it is not included in the list.

The amount of solutions that needs to be found is limited due to the factors that people consider important when choosing an operating system. It was identified from the background study that the solution needs to be widely spread to not be ruled out, since being widely spread has impact on many things; platform support, community support, availability of pre-built sources which often are more important factors than for example performance.

Further a solution that have thousands or millions of users will be put under an enormous amount of different conditions, providing trust in that it is stable and reliable. Anyone can make a real-time modification and provide benchmarks that shows promising performance, but without the wide spread use the trust in the predictability and stability will not be high enough. Theoretical guarantees can't alone provide enough trust for these complex systems.

With this in mind, when a possible solution is found, a Google search is made to see if there it is any official website for it. If the official website states that their solution doesn't have support for ARM (which BBB and RPi uses) then it wont be included on the list. If there is no official website or community website then the solution is excluded based on the discussion above. Also if the solution is not open source it will be excluded as well. The sources for information is not limited to published academic literature, but also includes official websites, forums and technical articles.

To be efficient the search method has been shaped so that it with good confidence captures all the possible candidates and only excludes solutions that wouldn't make it anyway. Google Scholar and Scopus has been used for this study.

Scopus
Search term: *(TITLE-ABS-KEY ("real-time") AND TITLE (linux))*

The first 50 hits sorted by both date and citation count is looked at, first the title, then if needed the abstract. When a name for the solution can be identified the next hit is examined.

Scholar
On scholar the first two pages with hits are examined, 40 in total, sorted by relevance.

Search term: *"real-time" linux*

Additionally, inspired by the approach used in [116] has necessary coverage for finding candidates to the optimal solution been achieved by using seminal papers as a base to find other sources. Like the first implementation of real-time Linux [117], any paper that attempts to make Linux real-time are likely to cite this one. However, some of them won't, but these solutions are then likely to be captured by looking at citing sources for the papers that originally cited [117]. If the original paper for a certain solution is found, then the first 15 citations of that paper will be looked at.

Lastly, some papers also have lists of Linux solutions. These types of papers are found by combining some of the seemingly poplar solutions in one search term (xenomai rtai linuxrk linuxrt) and (xenomai rtai preemp rt) on Google Scholar.

The candidate solutions that where found and made it to the list were: PREEMPT_RT, Xenomai, RTAI, Linux/RK, SCHED_DEADLINE, RTLinux, ART Linux, QLinux, KURT, Litmus.

4.1.3 Step 3. Refined Study

This part of the study aims to find literature that provides more detailed knowledge about the solutions. The question to be answered are: How have the real-time features been achieved, for example by using a single or dual kernel approach? How are the real-time features accessed, is there a special API? How widely spread is the solution, how good is the documentation? What level of support is there for popular single board computers like the BeagleBone Black or RaspberryPi, can you get it running with a few clicks or does it require a kernel expert? Who develops it, is there an organization behind it? The goal is to gather enough information to be able to either exclude or include the solution before the final step.

To determine how widely spread the solution is, the search count on Google Scholar and Google is used. Documentation and implementation details is found by looking at the official website.

The original papers of each solution are used to understand the purpose of the solution, the benefits, what it improves compared to previous ones. To find these papers the name of the solution was used as search term on Scholar and Scopus, sorting by date with the oldest first. Further, any paper of decent quality includes a reference for a solution if they discuss it, it could be a paper that has made an

implementation using for example Xenomai, by looking at the references the relevant paper can be found.

Beaglebone Black and Raspberry Pi support was determined by using Google, combining the name of a solution and the hardware, for example *xenomai raspsberry pi*. Community literature was found by Googling on the solution, looking for wikis and forums.

This information was used to reduce the list to three solutions: Xenomai, RTAI and PREEMPT_RT.

4.1.4 Step 4. Focused Study

For this final step sources where gathered to provide additional information in order to evaluate the solutions against the factors in Table 2. The main focus here is performance, Ada compatibility and real-time capabilities (Predictability).

Sources of information for Ada compatibility was found by searching on Scholar for Ada in combination with the solution, example : *ada xenomai*, and by finding the implementation details for the real-time features by looking at official websites.

The performance is an important aspect, to evaluate the solutions Scholar and Scopus were used to find benchmarks that compared the solutions. Since hardware plays an important role for the results, it was necessary that the studies included at least two of the solutions at the same time to evaluate how the perform against each other. If enough benchmarks are gathered a cross evaluation can be used. For example, one paper studies solution A and B, with the result that B is faster than A. Another paper studies solution B and C, where C is faster than B. Then it is reasonable to conclude that C is faster than A. The approach was therefore to use search terms as for example *xenomai rtai performance* and *xenomai preempt_rt performance*. The top sources sorted by date and citation count were first looked at. Then the looking at the citations to see if anyone had challenged their results. After reading a few papers it became clear that the results differed significantly depending on test methods and software versions, but it was still possible to see some trends. Based on that, and since performance was only one of many factors used for determining the optimal solution, enough sources where gathered to only get a rough estimation of how the solutions performed.

Sources used for evaluating the predictability factor where the official websites,

original papers that described the implementation, and sources found by searching on Scholar and full text search on IEEE for the solution in combination with the keyword predictability, for example *xenomai predictability*. The information found in this study was then used in Chapter 5 in order to determine the optimal one.

4.2 Programming languages

To find out how Ada compares to other languages in different situations Scholar, Scopus and full text IEEE was used to find sources discussing advantages and disadvantages. The search terms that where used were a mix of keywords: *Ada, C, Java, C++, programming languages, safety-critical, real-time, market studies, language selection, advantages, teaching, challenges*, further the references and citations where looked at to see if any interesting source could be found that either challenged or supported the view. Based on the quality, amount of new information, controversy of the subject a judgement call was made to determine when sufficient amount of sources had been collected. It resulted in quite a few sources which was difficult to keep track of, so important parts from each source was summarized into a document, see Appendix C, which simplified the overview. Then an analysis could be made, which can be found in Chapter 2.

5 Exploring Solutions

5.1 Mainline Linux

The default/mainline/vanilla Linux is too unpredictable regarding temporal aspects to be considered an RTOS. Instead predictability has been sacrificed for higher throughput. But why is there a trade off between these two? Consider the case of having multiple tasks (threads) that needs to execute simultaneously. A single processor (with one core) can only execute one task at hand, in order to show a seemingly concurrent behavior the the tasks needs to be scheduled. There are many different algorithms to schedule tasks and the amount of context switches between them will differ. The context switch is one of the most significant overheads in operating systems [91] and it's the most important speed issue in RTOSes [113]. The more switching between tasks means more lost CPU cycles, thus a GPOS can let a task run for longer until it finishes its job, or at least a big portion of it, reducing the amount of context switches, but this also means unpredictable latencies. For example in [118] it was shown that real-time patches for Linux resulted in 5 times more context switches compared to vanilla Linux.

A very important aspect of operating systems is the stability. Unexpected faults due to bit flips in CPU registers or corrupted API calls is a growing problem. With transistors becoming smaller and smaller the risk of faults increases because of higher sensitivity towards environmental radiation and component degradation [119]. Developers of operating systems needs to take this into consideration otherwise the OS won't be stable when put into a less protected environment.

Open-source OSes are just as reliable as proprietary ones, in [110] are three popular RTOSes are evaluated for these faults, µC/OS, VxWorks and RTEMS, of which the last one is open source. They use a system that corrupts data in for example API system calls and then observe how this fault is propagated through the system. The results shows that RTEMS is the most robust system according to their metric. They did not include Linux, but other papers have evaluated Linux in terms of fault injection which shows that Linux is a dependable system [120] and in [121] Linux performs well against MINIX, an OS running on a microkernel designed to be highly reliable [122].

Linux is a stable OS but in the vanilla kernel there are several things that makes it unsuitable as a RTOS, for example kernel system calls are not preemptible (can't be interrupted by other processes), paging of memory has no upper time bound, the default scheduler uses an unpredictable 'fairness' algorithm [111]. There are at least two main different ways in how to make Linux more predictable, either

by modifying the kernel to make it fully preemptive or by using a two-kernel approach [111]. See Figure 9 for an overview of the dual kernel approach used by the real-time extensions Xenomai and RTAI.

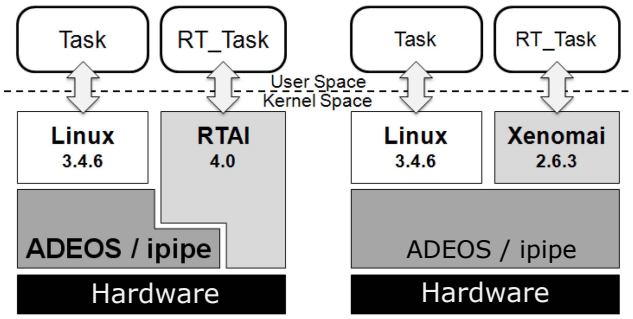

Figure 9: The dual kernel approach by Xenomai and RTAI to provide Linux with real-time capabilities. Linux runs as a low priority thread, interrupts are first delivered to the real-time kernel that then processes or redirects them. Figure from [123].

5.2 Real-Time Linux

The first real-time extension to Linux was created by Victor Yodaiken according to Butazzo [91], however according to [124] it was first implemented in the master thesis work of Barabanov [117] at the New Mexico Tech university where Yodaiken was a professor. In any case it went under the name RTLinux, then commercial available via FSMlabs and later on Wind River Systems, but now it is discontinued by Wind River which uses the PREEMPT_RT patches [125]. There was also a free RTLinux during the years which now seems to be abandoned. The way to improve the real-time behaviour was to use the dual kernel approach where Linux is running as a low priority preemptible thread, this has the benefit of low latencies, but also means that device drivers and other code has to be rewritten in order to be executed by the real-time kernel [91].

Similar approaches that exist today are RTAI [126] and Xenoma i[127], which provides performance at the same level as genuine RTOSes [128, 129]. They however suffer from the same drawbacks as RTLinux, code needs to be written using their API. Also, if for example code is written for Xenomai then great care needs to taken so no system calls are made to the Linux kernel since this would move the application to Linux space [130], resulting in lost real-time capabilities. The other approaches to modify the kernel itself is done by for example PREEMPT_RT patches and Linux/RK [131, 132].

Currently there are three real-time enchantments to Linux which has a history and long term feature that is consistently mentioned in the literature, PREEMPT_RT, Xenomai and RTAI [91, 118, 123, 133–140], to some extent there are implementations and papers about Linux/RK in for example [91, 103, 141]. The SCHED_DEADLINE [142] is now a part of the mainline kernel since Linux 3.14 [143]. In [139] are other options mentioned: ART Linux, QLinux and KURT. For scheduling in multiprocessor environments there is Litmus [140, 142, 144].

The PREEMPT_RT solutions modifies the kernel, the most important modifications are according to [139]:

- High resolution timers.
- Complete kernel preemption.
- Interrupts management as threads.
- Hard and soft IRQ as threads.
- Priority inheritance mechanism.

It is the most successful attempt to make the Linux kernel fully preemptive without using a dual kernel approach like Xenomai and RTAI [145]. Some sections like the scheduler and the implementation of mutexes must however remain non-preemptive [112]. On the other hand, a fully preemptive kernel might not be necessary, as [146] explains, a modern processor 1 GHz processor can execute 100,000 cycles in 100 μs, which means that the non-preemptive parts can execute fast enough to be sufficient for many applications. Also it is very difficult to account for the preemptive overheads in WCET analyses for fully preemptive systems [147].

Real-time systems does not necessarily mean fast systems, but what are the typical requirements? A market survey [49] on embedded systems asked respondents the about their timing demands, see Figure 10, which shows that often sub-millisecond performance is required, but for example the sample rate for a flight controller can be 100 ms [148]. According to [89] the requirements on response time for mechatronic systems are typically less than 100 μs. This means that if the RTOS can only provide a cycle time of 1 second there are many applications it can't be used for.

5.3 Ada Compatibility

Of the solutions mentioned above the only one that can be used with Ada without modification to the compiler is the PREEMPT_RT patches since it modifies the mainline kernel without requiring a special API to access the real-time functions: *The great advantage of PREEMPT_RT based solutions is that improved per-*

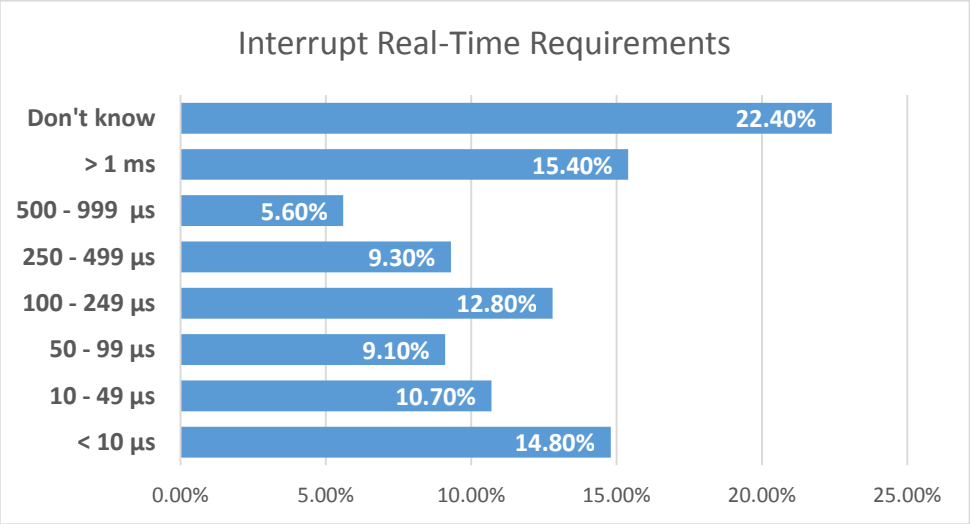

Figure 10: The real-time requirements for interrupt latencies, from a survey done by [49]

formance can be obtained instantly without any modifications of users code [145]. The drawback compared to Xenomai and RTAI is the lower performance in terms of latencies.

Ada can be used with RTAI or Xenomai without modifying the compiler, but then API bindings are needed to use those features. This is possible to do, but it means that the Ada features for managing for example tasks won't be used. There are already Ada bindings [149] to the the POSIX standard that many RTOS like Lynxos, VxWorks and Xenomai are conformant to, but as noted by [29] this means that that the Ada features won't be used and the portability will not be as good, it should only be used when there is no other option.

5.4 Candidates

Based on the factors in Table 2 there are three real-time modifications of Linux that stands out from the others:

- PREEMPT_RT
- Xenomai
- RTAI

These three options will be further investigated in this section, a comparison can be seen in Table 4.

Table 4: Factors that are considered for determining the optimal solution and how the different solutions scores.

Factors	PREEMPT_RT	RTAI	Xenomai
Ada compatibility	High	Low	Low
Performance	Medium	High	High
Portability	High	Medium	Medium
Predictability	Medium	High	High
Ease of use	High	Medium	Medium
Organization	High	High	High
History of use	High	High	High
Installation	High	Low	Low
Community	High	High	Medium
POSIX compliance	Medium	High	High
Documentation	High	High	Medium

Determining the best performance based on data from the literature can be difficult due to different methods and software versions used by the researchers. Garre et al [133] tests several real-time extensions, see Table 5, which are clearly showing Xenomai as the best performing. These results are completely different compared the ones in [129], which they say is probably due to two reasons, different benchmarking techniques and different versions. They give a reference to a more recent paper [150] that shows figures similar to theirs, RTAI is not included in that paper but Xenomai clearly outperforms PREEMPT_RT Linux when it comes to interrupt latencies. However, in [137, 151] experiments shows that RTAI have better performance than Xenomai in some applications.

Table 5: The worst case latencies in nanoseconds for Linux and it's real-time extensions, measured in latencies. Xenomai provides the shortest latencies. Data from [133].

	Task Switch [ns]	Preemption [ns]	IPC [ns]
RTAI	12060	14855	12986
Xenomai	**1422**	**2922**	**1282**
PREEMPT	27085	15986	19776
Linux	13200	$>10^6$	12453

In [130] Xenomai is compared to vanilla Linux when it comes to video decoding, it shows that stressing the system has a great impact on the performance. When the Linux system is being stressed the decoding times increases more than 50 times, but the Xenomai performance is remaining at the same level as for the unstressed system. Their conclusions is: *The Xenomai real-time extension has proven its high performance level of compliance with hard real-time constraints.*

Throughput benchmarks is provided by [139] where PREEMPT_RT is compared to vanilla Linux and Xenomai using an overall throughput benchmarking suite, the conclusions is that Xenomai is slightly better in a uniprocessor environment, however for a dual-core processor the mainline kernel scores highest.

When it comes to Xenomai vs PREEMPT_RT the recommendation in [150] is to use Xenomai for 100% hard real-time applications, but to consider the use of Linux with patches for 95% hard real-time application. However this was several years ago, the version they used, 2.6xx, is compared to 3.6xx in [152], which shows that for example the maximum interrupt latencies are almost 70% higher for the older Linux version. The difference between versions is even bigger for the task switch time and inter-process communication latencies, more than nine times larger [135]. Nevertheless, the PREEMPT_RT patches still is far from the performance of dedicated RTOS operating systems, but the gap have been reduced over the years [125].

A very important aspect is pointed out in [150]: "*In all likelihood, nobody is distributing pre-built, patched kernels for your platform, so using RT or Xenomai configurations commits you to maintaining a custom kernel in-house. Building custom-patched kernels is time consuming, and you will almost certainly encounter more bugs than you will in stock Linux. Identifying, classifying, and resolving those bugs is time consuming. Furthermore, since patches are developed and distributed separately, patch sets may lag the stock kernel head.*"

This aspect also mentioned by [85] that had problems of finding kernel sources for their UAV application using Linux/RK. However, today this is not such a big problem for the PREEMPT_RT patch that has received more attention and is long-term tested by OSADL [153] on many different platforms. They provide the config files being used for the tested kernels, making it possible to for anyone to compile a stable version, and the PREEMPT_RT patches are available through the official Linux website.

5.4.1 Hard Real-Time

The timing properties is essential for determining if an RTOS can be used at all for a certain task. Altenberg [154] investigates the PREEMPT_RT patches on ARM9 and ARM11 platforms. He concludes that cyclic tasks for these platforms can be used for cycle times down to 1ms and 0.5ms (this with the demand that the worst case latencies are a maximum 25% of the cycle time). As an comparison OMRON PLCs operates with cycle times from 0.15 ms to 1 ms depending on the model [155].

Altenberg makes a rather bold statement based on the test results: "*The benchmarking results on both platforms have proven a 100% deterministic behaviour of Preempt RT on ARM. So, Preempt RT on ARM CPUs can meet hard real-time requirements*". On the other hand in [145] tests are performed on ARM running vanilla Linux, PREEMPT_RT and ADEOS (the technology Xenomai and RTAI is built on), their conclusion is the opposite regarding PREEMPT_RT: "*In the group of evaluated solutions only the ADEOS based is suitable for hard real-time applications. PREEMPT_RT cannot guarantee the required determinism because of its architecture.*"

According to the previous discussions in this thesis about real-time it seems odd to categorize a RTOS as soft or hard based on data from benchmarks. For example in [3] it is said that the latencies of common Linux distributions can be hundreds of milliseconds and it is therefor not suitable for hard real-time applications. But what if there is an application that only require latencies in the order of seconds? Even if the latencies for the OS are measured in minutes it can still be a hard real-time operating system, there only need to be an upper bound on the latencies. Of course, if the latencies are very high and variable the RTOS can't be used for many applications.

In [156] Linux, PREEMPT_RT and Xenomai are benchmarked, one of their conclusions is that vanilla Linux is not suitable for real-time systems because the latencies under load scenarios are much larger than the mean values. But how large can the difference be before a system becomes unsuitable for real-time use? What makes Linux unsuitable for real-time systems cannot be shown by benchmarks on latencies, instead it is the theoretical analyses mentioned before that Linux has sections of unbounded latencies that makes it unsuitable for real-time systems.

Benchmarks cannot be used to guarantee that the upper bound of latencies has been found, at the most they can give information about the probability of how

likely is that no higher latency will occur. Even that is difficult, the probabilistic methods for timing analyses on complex hardware based on measurements (MBPTA) have strict requirements on the hardware. This approach can be a promising alternative to be used for complex hardware and software, but currently there is no commercial processor that is compliant with requirements for MBPTA [101]. Monitors, or Run-time Verification, is presented by [157] as an alternative, they say state-of-the art formal-verification can only on be used on a fraction of today's large systems and testing alone is impossible for ultra-reliable systems [148].

The only formally verified OS kernel with memory protection is the seL4 according to [158], the official web page [159] states that its the only OS with memory protection that can give hard real-time guarantees. However, the formal proofs only shows that the OS is functioning correctly according to the specification, but as [160] points out, formal methods shifts the problem of writing correct code to writing correct specifications. If the specification is not correct then the OS won't be either, so the statement that seL4 it can guarantee upper bounds is questionable.

The PREEMPT_RT patches for Linux modifies the unbounded sections to have upper bounds, but it is not possible to determine with 100% confidence for what conditions it is hard real-time. In the end it is up to the developers to conduct the necessary test to provide enough confidence that the tasks will meet their deadlines. For safety-critical applications requiring certification it will be very difficult to convince the certification authorities that meeting the deadlines can be guaranteed with high enough probability, one other option is then to use simpler hardware, with or without an simpler RTOS, that can provide higher confidence. Another option is to have a small section of code that does run-time verification and detects error [157], this code can them be formally verified but this approach won't work if the code runs on an operating system that itself cannot be verified. In [42] they looked at Linux compliance to the DO-178B standard but could not found any evidence that it had ever been done. There are however work done into certifying Linux, the OSADL [153] has a project, SIL2LinuxMP, that aims to certify Linux to comply with IEC 61508 SIL 2.

5.5 Summary

The optimal real-time solution for Linux depends entirely on the application, if one wants to program portable Ada code for Linux without bindings to Real-time APIs the only way to go is using PREEMPT_RT. If the real-time performance is

not sufficient, and full Ada support is not required, Xenomai or RTAI can be used. However, based on real-time requirements seen in Table 5, Xenomai and RTAI will not be sufficient for many systems either. The best option in many cases could therefor be to use Linux with PREEMPT_RT patches in combination with a separate micro-controller or similar for very low latency hard real-time tasks. This can be programmed in Ada if the platform has compiler for it, ARM STM32 and AVR32 are two architectures that can be used. The hard real-time demands shouldn't however be exaggerated, many systems today benefits more from a low average response time than a low maximum response time. For cars there are functions like Lane Keeping, Automatic Cruise Control and Crash Preparation are developed without hard deadlines, instead the average response time is the key quality parameter [161]. The Ada/PREEMPT_RT solution is well suitable for those types of applications.

6 Verification and Evaluation

The solution that will be furthered examined is the PREEMPT_RT patches running on a BeagleBone Black (BBB). The BeagleBone Black has been chosen because it is a popular board that is being long-term tested by OSADL. The requirement that the board running Linux should be tested by OSADL or any other organization is due to the fact the the Linux kernel can be configured in a million different ways, it is hard to tell how a small change will affect the system.

By using the exact Linux configuration tested by OSADL there is a lot more confidence in the stability of the system. Another option could be Raspberry Pi (RPi), but the test-reports on OSADL shows that it is not stable, this does not mean that it's impossible to have a stable real-time Linux version running on RPi, but no reliable information have been found for such a configuration.

The Linux kernel is only a part of the complete system, the device drivers that are developed by third parties runs in kernel space, if they are unstable it will make the system unstable. The real-time system is not just a matter of making the kernel real-time, the drivers have to work on that kernel without modifications for the solution to be considered successful.

The crashes of the RPi are probably due to the drivers, but the RPi2 does however show stable performance so far on the OSADL, it could be a option. The advantage for the BBB is its programmable real-time units (PRU) integrated on the board, directly accessible from the CPU. This means that no extra microcontroller would be needed for 100% hard real-time applications which would be the case for the RPi's.

The methods to evaluate the solution are:

- Implementing the modifications, installing it on a BeagleBone Black.
- Verifying the Ada support by testing the compiler using the GCC Ada testsuite.
- Executing benchmarks for evaluating real-time and throughput performance.

Typically, when benchmarking a real-time system, tests are done to measure the latencies, either with the system under load or not. However, as the literature study of hardware architectures have shown, there are many aspects of a modern microprocessor that affects the performance. The hypothesis here is that different types of applications might affect the latencies to a varying degree due to

cache, pipelines and branch predictions.

Instead of just loading the system in some arbitrarily way, the method here will be to use popular throughput test-suites for stressing system. For each test in the test-suite the latencies will be measured, as well as the throughput performance. This will show both how a high-priority task affects the throughput, and how the latencies of the high-priority task is affected by low-priority applications. It will show whether the system is sensitive to a certain type of workload. If there are significant latency differences when running for example a low-priority CPU application compared to a memory intensive one, then it's possible to optimize the application to for example use less memory operations in order to increase the performance of the real-time tasks.

6.1 Compiling the Kernel

One of the reasons for choosing a popular SBC as hardware platform was that is would make it more likely to find resources, tutorials etc on how to actually get a patched Linux kernel running on it. As mentioned, OSADL provides the kernel configuration, but for people that are not compiling kernels on a regular basis it is not very straightforward to go from there to a complete operating system running on the SBC. Fortunately after searching the web some scripts [162] where found that supposedly would leave you with a patched Linux operating system on a SD-card, ready to plug in to the BBB, by just issuing a command from the terminal.

However, as usual things never work 'right out of the box', first there where problems to get the cross-compiler working. Debian 8 was used in the beginning to execute the scripts, but after some unsuccessful effort with trying to install a cross-compiler and get it recognized by the script a switch was made to Lubuntu. From there all the dependencies could be installed by simple *apt-get install* commands.

Still, the scripts were not successful, there was no functioning OS on the SD-card after running them. Therefore the painful task of analyzing script line by line was needed to figure out was was wrong and how it could be fixed. This meant that significant time was spent on finding bugs instead of evaluating the system. At last the operating system got successfully compiled and transferred to the SD-card, the problem was faulty paths for the kernel modules when trying to transfer them to the SD-card. Unfortunately the problems where just starting, the OS was constantly freezing, errors where reported in the console when trying to simple tasks, it was completely unstable. After some more bug searching

it was realized that the SD-card had been partitioned to just fit the operating system, leaving no space left on the SD-card. Alot of the problems was solved by extending the rootfs partition into the unallocated space of the SD-card.

However, the freeze lags were still present, very high latencies were reported by cyclictest [163], frequently over 50ms. This caused confusion whether the PRREMPT_RT was properly installed or not, but *uname -a* showed that it was. The latencies and freezes finally got resolved by changing the CPU frequency scaling method to 'performance mode' which puts the CPU frequency to max, 1000 MHz, instead of 300 MHz that was the case for the default 'ondemand' policy. However, this was later silently without notification changed back to default when rebooting. By then many packages had been install, the thought was that any of these were causing the problems, it took a while to realize that the scaling method had been changed and were the cause of problems.

6.2 Verifying the Compiler

The Gnat (Ada) compiler is not installed by default, this is however simply fixed by issuing *apt-get install gnat* in the terminal. This will install the 4.6 gnat compiler and the AdaCore GPS development environment. More recent compiler version can of course be installed by either downloading the GCC sources and building them or by updating the source-lists for apt-get to include later Debian packages.

To test the correctness of the compiler the GCC testsuite was downloaded from https://ftp.gnu.org/gnu/gcc/gcc-4.6.3/gcc-testsuite-4.6.3.tar.gz

Dejagnu is a prerequiste for the GCC testsuite so it was installed by *apt-get install dejagnu*

The first attempt to verify the compiler by executing *make check-ada* in accordance with the GCC guidelines (https://gcc.gnu.org/onlinedocs/gccint/Ada-Tests.html) just resulted in: *make: *** No rule to make target 'check-ada'. Stop.*

After some web-searching and trial-and-error the solution was to issue *runtest –tool gnat*, executed from the testsuite folder. This did however not work as expected, there were a lot of errors. Extending the command, with the *-v* option, showed when the failures occurred and gave some more information : *RTS path not valid: missing adainclude and adalib directories.* Trying to solve this by modifying the paths, (ADA_INCLUDE_PATH, ADA_OBJECTS_PATH), to point

directly to the adainclude and adalib folders did not help. After some more web-searching on DejaGnu (driver for executing the tests) it was found that more detailed output could be achieved by just adding more *-v's*:
runtest –tool gnat -v -v

This produced a lot of output, and in the noise this was found: *–RTS=/usr/./libada*. That is the path the compiler is using, so the solution was then to simply create that folder and copy adainclude and adalib there:
cp -r /usr/lib/gcc/arm-linux-gnueabihf/4.6/adalib/ /usr/libada/

This solved the problem and the test could finally execute as intended, results were:
=== gnat Summary ===

# of expected passes 868
# of expected failures 11
# of unsupported tests 5

This shows is that the compiler works as intended, there where no unexpected passes or failures. The five tests that where unsupported were targeted for other platforms, like the test for SSE instructions used on the x86 architecture.

6.3 Benchmarking

A popular software tool to benchmark real-time performance on Linux is cyclictest [163], the benefits of testing the performance without using external equipment like an oscilloscope and a function generator is the simplicity of setting up repeatable tests. The downside is that the response latencies for external events triggered by IO wont be measured, but as shown in [154], cyclictest shows similar figures to tests that toggles GPIO pins and measures the latencies. It is therefore possible to draw conclusions based on the tests here on how the system would behave if external interrupts are enabled on the IO pins for the BBB.

The system was loaded by runnin hackbench [164], parts of SysBench[165] and UDP Unicorn (ping flood) [166], see Table 6. At the same time where cyclictest running as a high priority process and measuring the latencies. The purpose of this setup is to stress many parts of the system in a somewhat isolated way to see if there are types of applications that influence the real-time performance more than others. A RTOS where a high-priority task is sensitive to applications running at a low priority is not very predictable and in a sense not suitable for

hard real-time systems. The opposite applies for a GPOS that should be 'fair' when dividing the resources amongst applications, achieving high throughput. In that case it is not desirable that a single application consumes all the resources and severely reduces the throughput of the systems just to maintain low latencies for itself.

Table 6: The tests used for benchmarking and stressing the system

Test name	Description
CPU	SysBench, CPU intensive.
Memory	SysBench, memory intensive.
Threads	SysBench, stresses the scheduler.
FileIO	SysBench, file I/O intensive.
Ping	UDP Unicorn, system stressed by ping flooding.
Hackbench	RT-Tests, stresses the system by sending data between threads.
Cyclictest	RT-Tests, benchmarks the latencies.
No Load	The state after system reboot, no other applications started.

There is no global optimal solution in how response time and throughput should be prioritized, it is entirely dependent on the application. The take away is that the operating system should be customizable, the user needs to have the option to choose to what extent different aspects should be prioritized. Linux provides this by having different schedulers for both real-time and general-purpose applications that can be controlled by the user.

6.3.1 Benchmarking Setup

The relevant software for conducting the tests are:

- SysBench 0.4.12
- RT-tests 0.92 (cyclictest and hackbench)
- Linux 3.12.31-rt45
- UDP Unicorn 2.0

See Appendix A for details about setting up the software.

Each test was configured to run for approximately 10 minutes, this was judged to be sufficient to determine if the latencies where affected in any significant way by the load.

Bash scripts where made to test and log the system in a systematical way, see Appendix B. For all the tests, except the ping flood, the BeagleBone Black was connected to a Logitech K400 keyboard/touchpad and a Samsung monitor. The power was supplied via a 2.1A 5V power supply (DC connector) since powering the BBB via USB at 500 mA have been reported to cause problems. For the ping flood test a Ethernet cable was connected, via a TP-Link MR-3420 router, to a quad-core Intel computer. The only command issued before running the test after reboot was to set the CPU governor to run in 'performance' mode. The tests where then started from the terminal on the BBB.

6.4 Results and Discussion

The results, as seen in Figure 11, shows some variation between the different workloads. One test that stands out is the ping flood, generating worst case latencies that are are 3-5 times larger compared to the other loads, see Figures 13 and 14 for distribution plots. When loading the system with the Thread-test are the latencies surprisingly lower compared to the unloaded system. Running two more tests to confirm this, the Thread-test showed lower latencies compared to the No-Load case. This either shows that the Thread-test have a unexpected influence on the system or it's just a fluke. Other results that is worth mentioning is that during the ping-flood the GUI completely frooze, something that needs to be considered if the BBB is hooked up to the Internet.

The results of Thread-test are indeed strange, logically an unstressed system would show lower latencies than a stressed one. However, remember the discussion about timing anomalies, where the the trigger of a slow cache miss can results in lower total executions times compared to the much faster cache hit. The Thread-test might be one of these odd cases where the complex architecture and software is combined in a way that make it difficult to reason about what to expect without rigorous analyses down to the bit level. This is something that in future work should be investigated in more detail to determine the cause.

The differences between the other tests CPU, Memory and Hackbench can be considered to be insignificant, in the sense that a developer doesn't need to deploy advanced techniques to for example reduce memory operations or file IO. Such optimizations won't affect the latencies for the high priority task significantly, but is likely to increase development time and cost. Further, between the CPU and Memory-test is the difference less than 6%, with the CPU-test showing the lowest latencies, if the test where to run for a week it is not unlikely that this relation could be swapped. If the BBB is connected to the Internet a developer

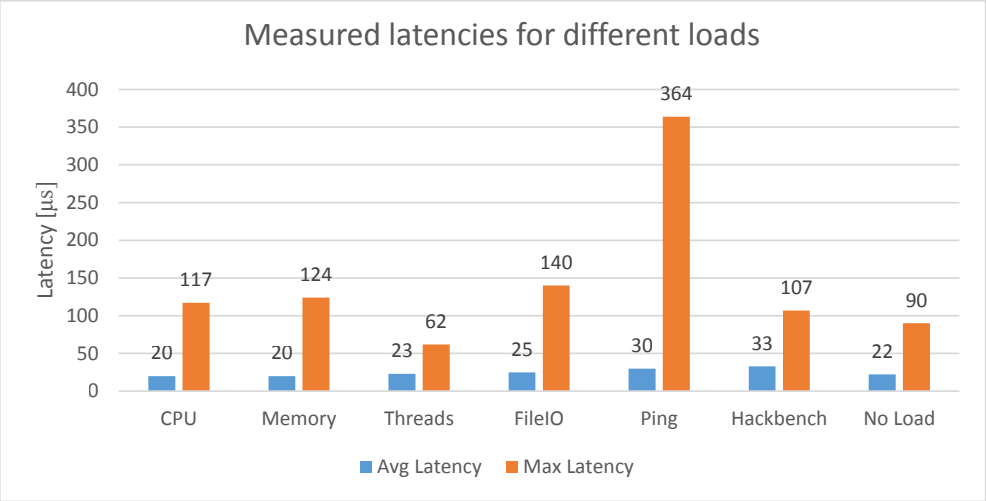

Figure 11: The latencies under various kinds of loads on the system. The two tests that stands out are the Ping and Threads tests. Perhaps most surprising is that the Threads-test load actually results in lower the latencies for the cyclictest task, it is unclear why this is the case.

needs to have in mind that the worst case latencies will be several times higher, for this case it is motivated to use strategies for handling packet flow to the BBB, perhaps by using a dedicated firewall on another hardware platform.

The throughput performance got significantly affected by the cyclictest task, around 20-30% for some tests, see Figure 12, as a reference the cyclictest used about 10% of the CPU, reported by the *top* command. If this amount of performance degradation is acceptable is dependent on the application of the system, but considering that the cyclictest used 10% of the CPU a 20-30% performance decrease for the low priority tasks is not extreme.

The research was designed with a strong focus on the literature study, this to have a good understanding of the requirements for different types of system, e.g soft/hard real-time and how the complete platform Linux+Ada+SBC meets those requirements. However, this reduces the amount of time that can be used for testing and analysing the platform. If the background study would have been reduced more time could have been spent on interpreting and investigating the benchmark results, for example the low latencies for Thread-test.

The parts that could have been reduced without severely negatively affecting results and conclusions could have been the work of finding and narrowing down

the possible Linux real-time solutions. Instead of looking at all possible solutions, PREEMPT_RT, Xenomai and RTAI could have been focused on from the beginning, although this would have changed the formulation of the research. Instead of finding the optimal solution it would have been an evaluation of three popular solutions. Depending on who reads this report this could be both positive and negative. For someone who is well addressed with the different types of Linux real-time solutions, a stronger focus on testing could have been more desirable. For others, the effort to consider all solutions is necessary to provide confidence that the most promising solution have been tested and analyzed, so that a decision whether to use Linux+Ada or not is based on the state-of-the-art and not some less promising solution.

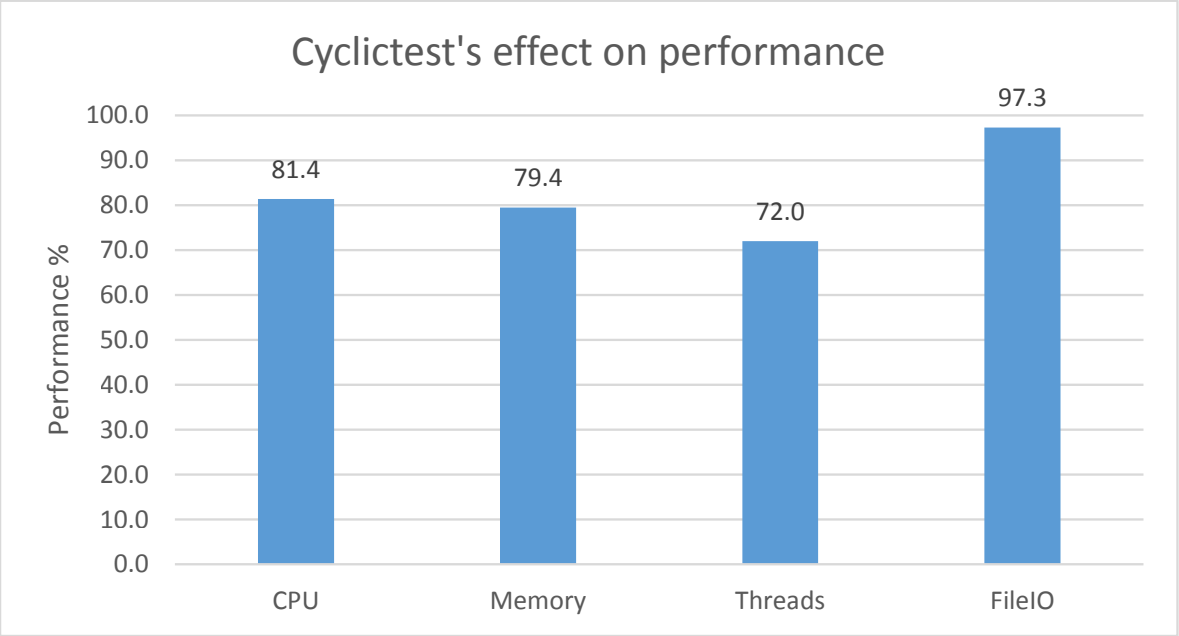

Figure 12: The performance for various SysBench tests when running Cyclictest at the same time, compared to the case when the system is not loaded. The FileIO is almost not affected at all, for the other tests there is a significant decrease in performance. The cyclictest used 10% of the CPU when running the tests.

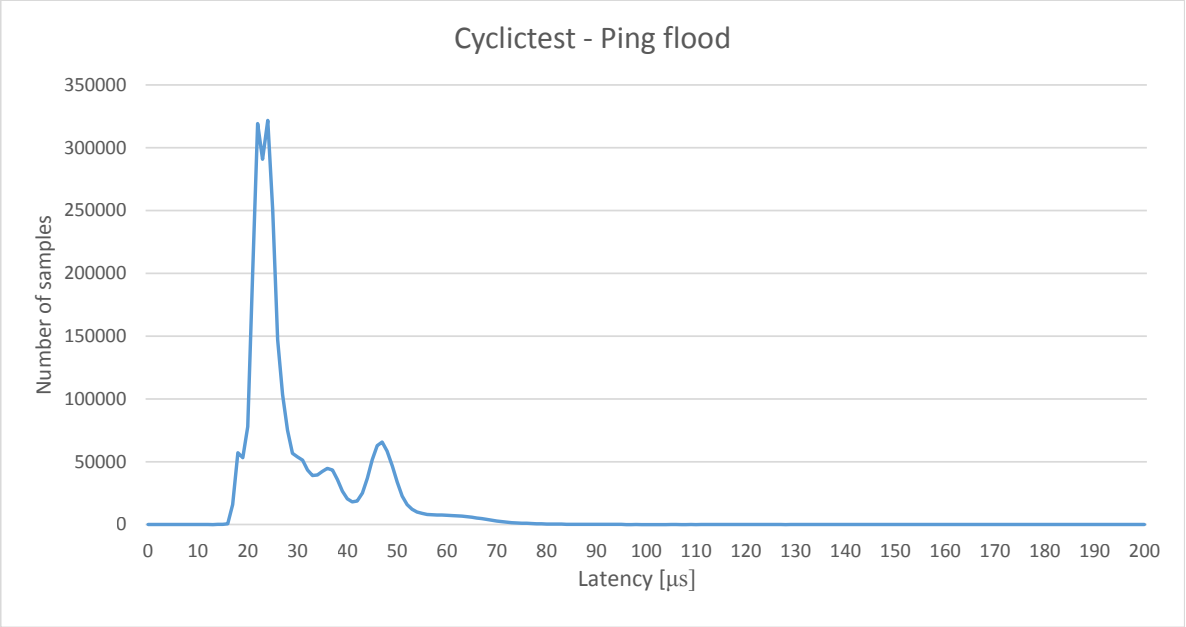

Figure 13: Latencies reported by cyclictest with ping flood as load.

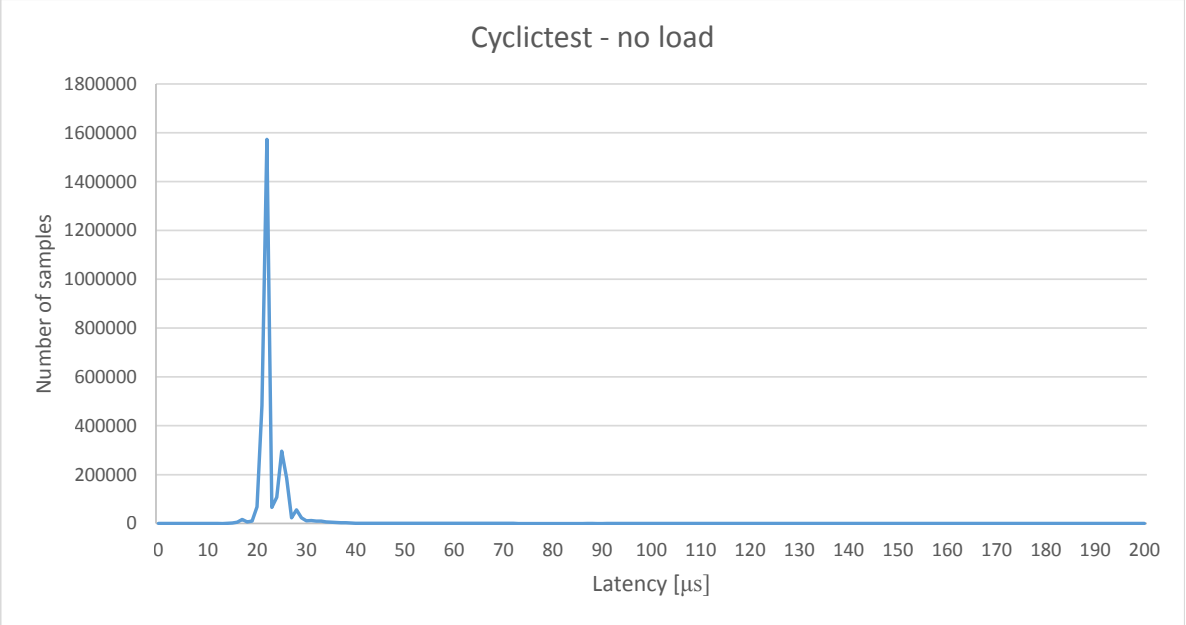

Figure 14: Latencies reported by cyclictest without any load on the system.

7 Conclusions

The tests in Section 6 combined with the discussions in Section 5 shows that Linux with Preempt_RT patches can provide predictable worst case latencies for systems where applications from many different domains are running simultaneously. Latencies are fairly consistent across various types of loads on the system, except when stressed from ping floods. One of the tests showed unexpected results, loading the system with the Thread-test resulted in lower latencies compared to the unloaded system. Future work that investigates why that happened could result in new knowledge regarding the real-time behaviour of PREEMPT_RT Linux.

The analysis have been limited to only include Linux real-time solutions that have received enough attention to be cited by others in literature. The practical part the thesis, evaluation and verification, was limited to the most promising solution, PREEMPT_RT patches running on an BeagleBone Black, which is the solution that answers the research question, see Section 3.2.

If the strict definition for hard real-time system is used - no deadlines are allowed to be missed - then Ada code cannot practically run in hard real-time on this platform. Some use real-time benchmarks to determine if the RTOS is hard real-time, but as discussed before this is highly questionable. Tests can never prove that the maximum latencies have been measured and full Linux is too complex to be verified using code analysis with current state-of-the-art methods. This also true for other RTOSes, it is not possible to 100% guarantee that the deadlines will be met except for the simplest hardware and software, but this is usually not a requirement. Certifications standards deals with probabilities, but that is also difficult to estimate for this Linux solution running on the BBB, except for the lowest safety levels where testing the system for a certain amount of time is feasible.

The general advise based on the findings is not to use Linux for the highest safety levels of safety-critical applications, other than that is well suited for mixed real-time general-purpose applications. The BeagleBone Black can provide hard real-time determinism with its PRUs, but the objective here was to see if Ada code can run on top of Linux, not a bare-metal microprocessor.

As discussed earlier, people developing real-time systems are often not experts on compiling operating systems, a suitable OS+platform configuration would not require this from the users. The solution evaluated here lacks in that aspect, it was not easy to get the system up and running. It would be very beneficial if

the people developing these hardware platforms would provide a pre-built PREEMPT_RT Linux distribution that only requires that the user clicks a few buttons. OSADL is testing PREEMPT_RT Linux on many different hardware platforms, they provide configuration files and patch scripts, but what really is needed in order to bring real-time Linux to the masses is to provide complete package that even the most inexperienced Linux user can download and get running quickly.

Programming and building Ada software on the BBB is possible without any other computer involved. This means that it's not necessary to find cross-compilers, which is a huge benefit. However, for large applications a cross-compiler will probably be needed that can run on a more powerful computer if the compilation times cannot be accepted. Cross-compiling is standard business in the world of Linux, so if one can't find one there are tutorials on how to make one.

The research goals, see Section 3.1, have been met by providing an analysis of Ada in section 2.1.1 that answers why, how, when Ada is used, its strengths and its future. The factors that are important when choosing operating systems can be seen in Table 2. The deeper understanding of why people choose a particular language and operating system leads to the conclusions that performance and technical features are not enough to motivate transitions from other solutions in most cases. Ada needs to become more widely spread, which can be done if its pushed into the curricula of universities. The new focus on pedagogical benefits rather than industry demand puts Ada at a better position than it has been in many years. The solution presented here, Ada+PREEMPT_RT+BBB is a cost-effective, readily available combination that could be used for that purpose, as well as for prototyping and applications that doesn't require the highest levels of safety certification. Finally, the approach of using one hardware platform for many different types of functions, instead of several platforms, is in accordance with environmental requirements of reducing energy consumption, it also reduces the stress on the environment related to factors from production, transportation and recycling of the hardware.

References

[1] C Ebert and C Jones, "Embedded Software: Facts, Figures, and Future", *Computer*, vol. 42, no. 4, pp. 42–52, 2009.

[2] M. Broy, M. V. Cengarle, and E. Geisberger, "Cyber-physical systems: imminent challenges", in *Large-Scale Complex IT Systems. Development, Operation and Management*, Springer, 2012, pp. 1–28.

[3] B. Bouyssounouse and J. Sifakis, *Embedded Systems Desgin: The ARTIST Roadmap for Reasearch and Development*. Springer Science & Business Media, 2005, vol. 3436.

[4] L. Sha, T. Abdelzaher, K. E. Årzén, A. Cervin, T. Baker, A. Burns, G. C. Buttazzo, M. Caccamo, J. Lehoczky, and a. K. Mok, "Real time scheduling theory: A historical perspective", *Real-time systems*, vol. 28, pp. 101–155, 2004.

[5] H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Springer Science & Business Media, 2011.

[6] B. Marco, D. Natale, A. L. Sangiovanni-Vincentelli, M. D. Natale, and A. L. Sangiovanni-Vincentelli, "Moving from federated to integrated architectures in automotive: The role of standards, methods and tools", *Proceedings of the IEEE*, vol. 98, no. 4, pp. 603–620, 2010.

[7] C. B. Watkins and R. Walter, "Transitioning from federated avionics architectures to Integrated Modular Avionics", *AIAA/IEEE Digital Avionics Systems Conference - Proceedings*, pp. 1–10, 2007.

[8] E. Lee, "Cyber Physical Systems: Design Challenges", *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, 2008.

[9] L. I. Millett, M. Thomas, and D. Jackson, *Software for Dependable Systems:: Sufficient Evidence?* National Academies Press, 2007.

[10] P. Marwedel, *Embedded System Design*, 2nd. Springer Netherlands, 2011.

[11] G. Buttazzo, "Research trends in real-time computing for embedded systems", *ACM SIGBED Review*, vol. 3, no. 3, pp. 1–10, 2006.

[12] P. R. Kumar, "Cyber–Physical Systems: A Perspective at the Centennial", *Proceedings of the IEEE*, vol. 100, pp. 1287–1308, 2012.

[13] R. Wilhelm and D. Grund, "Computation takes time, but how much?", *Communications of the ACM*, vol. 57, no. 2, pp. 94–103, 2014.

[14] J. Rogers, "Language choice for safety critical applications", *ACM SIGAda Ada Letters*, pp. 81–90, 2011.

[15] P. Liggesmeyer and M. Trapp, “Trends in embedded software engineering”, *IEEE Software*, vol. 26, no. 3, pp. 19–25, 2009.

[16] P. Derler, E. a. Lee, M. Törngren, and S. Tripakis, “Cyber-physical system design contracts”, in *Proceedings of the ACM/IEEE 4th International Conference on Cyber-Physical Systems*, ACM, 2013, pp. 109–118.

[17] P. A. Laplante and S. J. Ovaska, *Real-time Systems Design and Analysis*, 4th. IEEE Press, 2012, p. 529.

[18] UBM, “2014 Embedded Market Study”, Tech. Rep., 2014.

[19] S. Behere, F. Asplund, A. Söderberg, and M. Törngren, “Architecture challenges for intelligent autonomous machines: An industrial perspective”, in *13th International conference on Intelligent Autonomous Systems (IAS-13), Padova 15-19 July 2014*, 2014.

[20] B. Jacob, S. Ng, and D. Wang, *Memory Systems: Cache, DRAM, Disk*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[21] Coverity, “Coverity Scan : 2013 Open Source Report”, Tech. Rep., 2013.

[22] C. Rommel, “Brewing Embedded Market Success with Java”, Tech. Rep.

[23] S Girbal, M Moreto, A Grasset, J Abella, E Quinones, F. J. Cazorla, and S Yehia, “On the convergence of mainstream and mission-critical markets”, *Design Automation Conference (DAC), 2013 50th ACM / EDAC / IEEE*, pp. 1–10, 2013.

[24] K. Ramamritham and J. A. Stankovic, “Scheduling algorithms and operating systems support for real-time systems”, *Proceedings of the IEEE*, vol. 82, no. 1, pp. 55–67, 1994.

[25] R. Ma, W. Ye, A. Liang, H. Guan, and J. Li, “Cache isolation for virtualization of mixed general-purpose and real-time systems”, *Journal of Systems Architecture*, vol. 59, no. 10, pp. 1405–1413, 2013.

[26] J. Barnes, *Programming in Ada 2012*. Cambridge University Press, 2014.

[27] J. Barnes, *Ada Rationale 2012*.

[28] S. T. Taft, R. A. Duff, R. L. Brukardt, E. Ploedereder, P. Leroy, E. Schonberg, and D. Hutchison, *Ada 2012 Reference Manual*. 2012, vol. 2012.

[29] J. W. McCormick, F. Singhoff, and J. Hugues, *Building parallel, embedded, and real-time applications with Ada*. Cambridge University Press, 2011.

[30] A. Burns and A. Wellings, *Real-Time Systems and programming languages*, 4th. Addison-Wesley Educational Publishers Inc, 2009, p. 602.

[31] F. Singhoff, "Real Time Scheduling and its use with Ada", in *Tutorial presented to the annual international conference of the ACM SIGAda*, 2007, x–y.

[32] S. F. Zeigler, "Comparing development costs of C and Ada", *Rational Software Corporation, Santa Clara, Calif., March*, vol. 30, 1995.

[33] J. W. McCormick, "Ada and software engineering education: One Proferssor's Experiences", *ACM SIGAda Ada Letters*, vol. 28, p. 91, 2008.

[34] R. L. Glass, "Some thoughts on automatic code generation", *ACM SIGMIS Database*, vol. 27, no. 2, pp. 16–18, 1996.

[35] M. Domeika, "Programming Languages", in *Real World Multicore Embedded Systems*, First Edit, Elsevier Inc., 2013, ch. 9, pp. 289–312.

[36] P. K. P. Lawlis, "Guidelines for choosing a computer language: support for the visionary organization", *Defence Information Systems Agency*, pp. 1–14, 1997.

[37] D. Naiditch, "Selecting a programming language for your project", *IEEE Aerospace and Electronic Systems Magazine*, vol. 14, no. September, pp. 11–14, 1999.

[38] M. Nahas and A. Maaita, *Choosing Appropriate Programming Language to Implement Software for Real-Time Resource-Constrained Embedded Systems*. INTECH Open Access Publisher, 2012.

[39] G. J. G. Holzmann, "The power of 10: rules for developing safety-critical code", *Computer*, vol. 39, no. 6, pp. 95–99, 2006.

[40] J. a. McDermid, "Software safety: where's the evidence?", in *Proceedings of the Sixth Australian workshop on Safety critical systems and software-Volume 3*, vol. 3, Australian Computer Society, Inc., 2001, pp. 1–6.

[41] I. Dodd and I. Habli, "Safety certification of airborne software: An empirical study", *Reliability Engineering and System Safety*, vol. 98, no. 1, pp. 7–23, 2012.

[42] A. Kornecki and J. Zalewski, "Certification of software for real-time safety-critical systems: State of the art", *Innovations in Systems and Software Engineering*, vol. 5, pp. 149–161, 2009.

[43] R. Bell, "Introduction to IEC 61508", in *Proceedings of the 10th Australian Workshop on Safety Critical Systems and Software - Volume 55*, ser. SCS '05, Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2006, pp. 3–12.

[44] T. Kelly and J. McDermid, "Software in Safety Critical Systems: Achievement and Prediction", *Nuclear Future*, vol. 2, no. 3, pp. 140–146, 2006.

[45] B. M. Brosgol, “Ada 2005: a language for high-integrity applications”, *CrossTalk–The Journal of Defense Systems*, vol. 19, no. 8, pp. 8–11, 2006.

[46] B. Brosgol, “Safety and security: Certification issues and technologies”, *Crosstalk, The Journal of Defense Software Engineering*, pp. 9–14, 2008.

[47] B. a. Wichmann, “Languages for Critical Systems”, *Failure and Lessons Learned in Information Technology Management*, vol. 2, pp. 207–210, 1998.

[48] F. Wartel, L. Kosmidis, C. Lo, B. Triquet, E. Quiñones, J. Abella, A. Gogonel, A. Baldovin, E. Mezzetti, L. Cucu, T. Vardanega, and F. J. Cazorla, “Measurement-based probabilistic timing analysis: Lessons from an integrated-modular avionics case study”, *Proceedings of the 8th IEEE International Symposium on Industrial Embedded Systems, SIES 2013*, no. Sies, pp. 241–248, 2013.

[49] VDC, “Select Findings : Embedded Engineering Survey”, no. August, 2011.

[50] L. A. Meyerovich and A. S. Rabkin, “Empirical analysis of programming language adoption”, in *ACM SIGPLAN Notices*, vol. 48, ACM, 2013, pp. 1–18.

[51] IEEE Spectrum. (2014). Programming Language Ranking, [Online]. Available: `http://spectrum.ieee.org/static/interactive-the-top-programming-languages` (visited on Jul. 5, 2015).

[52] TIOBE. (2015). TIOBE Index, [Online]. Available: `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html` (visited on Jun. 30, 2015).

[53] A. A. Hook, C. W. Mcdonald, S. H. Nash, C. Youngblut, B. Brykczynski, C. W. Mcdonald, S. H. Nash, and C. Youngblut, “A Survey of Computer Programming Languages Currently Used in the Department of Defense.”, Tech. Rep., 1995.

[54] Y. Chen, R. Dios, A. Mili, L. Wu, and K. Wang, “An empirical study of programming language trends”, *IEEE Software*, vol. 22, pp. 72–79, 2005.

[55] B. Davey, K. R. Parker, B. Davey, and K. R. Parker, “Language History - A Tale of Two Countries”, in *History of Computing and Education 2 (HCE2)*, vol. 215, Springer US, 2006, pp. 139–151.

[56] K. R. Parker and B. Davey, “The History of Computer Language Selection”, in *Reflections on the History of Computing*, Springer Berlin Heidelberg, 2012, pp. 166–179.

[57] J. M. Zelle, “Python as a first language”, in *Proceedings of 13th Annual Midwest Computer Conference*, vol. 2, 1999, p. 145.

[58] R. E. Sward, M. C. Carlisle, B. S. Fagin, and D. S. Gibson, “The case for Ada at the USAF academy”, *ACM SIGAda Ada Letters*, vol. XXIV, pp. 68–70, 2004.

[59] R. E. Sward, “The rise, fall and persistence of Ada”, *ACM SIGAda Ada Letters*, vol. 30, p. 71, 2010.

[60] K. Lundqvist and J. Srinivasan, “A first course in software engineering for aerospace engineers”, *Software Engineering Education Conference, Proceedings*, vol. 2006, pp. 77–84, 2006.

[61] R. Mason and G. Cooper, “Introductory Programming Courses in Australia and New Zealand in 2013 - trends and reasons”, *16th Australasian Computing Education Conference*, pp. 139–147, 2014.

[62] D. Reifer, J. Craver, M. Ellis, and D. Strickland, “Is Ada Dead or Alive Within the Weapons System World”, *Crosstalk*, vol. 13, no. 12, pp. 22–25, 2000.

[63] M. B. Feldman, “Ada experience in the undergraduate curriculum”, *Communications of the ACM*, vol. 35, no. 11, pp. 53–67, 1992.

[64] T. Tempelmeier, “Teaching ‘ Concepts of Programming Languages ’ with Ada”, in *Proceedings of the 17th Ada-Europe international conference on Reliable Software Technologies*, Springer-Verlag, 2012, pp. 60–74.

[65] R. M. Siegfried, D. M. Greco, N. G. Miceli, and J. P. Siegfried, “Whatever Happened to Richard Reid’s List of First Programming Languages ?”, *Information Systems Educators Conference*, vol. 10, no. 4, pp. 1–7, 2011.

[66] A. Stefik and S. Hanenberg, “The programming language wars: Questions and responsibilities for the programming language community”, in *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, ACM, 2014, pp. 283–299.

[67] K. R. Parker, T. a. Ottaway, and J. T. Chao, “Criteria for the selection of a programming language for introductory courses”, *International Journal of Knowledge and Learning*, vol. 2, p. 119, 2006.

[68] a Pears, S Seidman, L Malmi, L Mannila, E Adams, J Bennedsen, M Devlin, and J Paterson, “A survey of literature on the teaching of introductory programming”, *SIGCSE Bulletin*, vol. 39, no. 4, pp. 204–223, 2007.

[69] S. Davies, J. a. Polack-Wahl, and K. Anewalt, “A snapshot of current practices in teaching the introductory programming sequence”, *Proceedings of the 42nd ACM technical symposium on Computer Science Education - SIGCSE ’11*, p. 625, 2011.

[70] L. F. Johnson, “C in the first course considered harmful”, *Communications of the ACM*, vol. 38, no. 5, pp. 99–101, 1995.

[71] R. B. K. Dewar and E. Schonberg, “Computer Science Education : Where Are the Software Engineers of Tomorrow ?”, *CROSSTALK The Journal of Defense Software Engineering*, no. January, 2008.

[72] J. L. Hennessy and D. a. Patterson, *In Praise of Computer Architecture : A Quantitative Approach*, 5th. Burlington : Elsevier Science, 2011, p. 857.

[73] M. D. McCool, A. D. Robison, and J. Reinders, *Structured parallel programming: patterns for efficient computation*. Elsevier, 2012.

[74] V. Huard, F. Cacho, and X. Federspiel, “Technology scaling and reliability challenges in the multicore era”, *IEEE International Reliability Physics Symposium Proceedings*, pp. 1–7, 2013.

[75] F. Sebek, September. Mä.

[76] S. Borkar and A. A. Chien, “The Future of Microprocessors”, *Communications of the ACM*, vol. 54, no. 5, pp. 67–77, 2011.

[77] G. Hager and G. Wellein, *Introduction to high performance computing for scientists and engineers*. CRC Press, 2010, p. 356.

[78] H.-S. P. Wong and S. Salahuddin, “Memory leads the way to better computing”, *Nature Nanotechnology*, vol. 10, no. 3, pp. 191–194, 2015.

[79] J. a. Stankovic, “Misconceptions about real-time computing: A serious problem for next-generation systems”, *Computer*, vol. 21, no. 10, pp. 10–19, 1988.

[80] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm, “The influence of processor architecture on the design and the results of WCET tools”, *Proceedings of the IEEE*, vol. 91, no. 7, pp. 1038–1054, 2003.

[81] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Pushner, J. Staschular, and P. Stenstrom, “The Worst-Case Execution Time Problem — Overview of Methods and Survey of Tools”, *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 3, pp. 1–52, 2008.

[82] Y. L. Y. Lu, T. Nolte, J. Kraft, and C. Norstrom, “A Statistical Approach to Response-Time Analysis of Complex Embedded Real-Time Systems”, *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2010 IEEE 16th International Conference on*, 2010.

[83] P. Axer, C. Rochange, M. Sebastian, R. V. Hanxleden, R. Wilhelm, W. Yi, R. Ernst, H. Falk, A. Girault, D. Grund, N. Guan, B. Jonsson, P. Marwedel, and J. Reineke, "Building timing predictable embedded systems", *ACM Transactions on Embedded Computing Systems*, vol. 13, no. 4, pp. 1–37, 2014.

[84] M. Schoeberl, "Time-predictable chip-multiprocessor design", *Conference Record - Asilomar Conference on Signals, Systems and Computers*, pp. 2116–2120, 2010.

[85] D. De Niz, L. Wrage, N. Storer, A. Rowe, and R. Rajkumar, "On resource overbooking in an unmanned aerial vehicle", *Proceedings - 2012 IEEE/ACM 3rd International Conference on Cyber-Physical Systems, ICCPS 2012*, pp. 97–106, 2012.

[86] P. Graydon and I. Bate, "Realistic safety cases for the timing of systems", *Computer Journal*, vol. 57, no. 5, pp. 759–774, 2014.

[87] J. H. Koh and B. W. Choi, "Real-time performance of real-time mechanisms for RTAI and Xenomai in various running conditions", *International Journal of Control and Automation*, vol. 6, no. 1, pp. 235–246, 2013.

[88] M. Hedlund and F. Aronson, "Evaluation of real-time operating systems for safety-critical systems", Jönköping University, 2002.

[89] J. Wikander and B. Svensson, *Real-Time Systems in Mechatronic Applications*. Springer US, 1998, 135 p.

[90] J. A. Stankovic and K. Ramamritham, "What is predictability for real-time systems?", *Real-Time Systems*, vol. 2, pp. 247–254, 1990.

[91] G. C. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer Publishing Company, Incorporated, 2011.

[92] P. B. Sousa, "Real-Time Scheduling on Multi-core: Theory and Practice", PhD thesis, Universidade do Porto, 2013.

[93] G. Bernat, "Specification and analysis of weakly hard real-time systems", PhD thesis, Universitat de les Illes Balears, 1998.

[94] W. A. Halang, R. Gumzej, M. Colnaric, and M. Druzovec, "Measuring the performance of real-time systems", *Real-time systems*, vol. 18, no. 1, pp. 59–68, 2000.

[95] A. Burns, A. Burns, B. Dobbing, B. Dobbing, T. Vardanega, and T. Vardanega, "Guide for the use of the Ada Ravenscar Profile in high integrity systems", *ACM SIGAda Ada Letters*, vol. XXIV, no. January, pp. 1–74, 2004.

[96] C. M. Krishna, *REAL-TIME SYSTEMS*, 1. Wiley Online Library, 1999, pp. 262–270.

[97] G. C. Buttazzo, "Rate Monotonic vs. EDF: Judgment Day", *Real-Time Systems*, vol. 29, no. 1, pp. 5–26, 2005.

[98] G. Bernat, A. Burns, and A. Liamosi, "Weakly hard real-time systems", *Computers, IEEE Transactions on*, vol. 50, no. 4, pp. 308–321, 2001.

[99] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand, "Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 7, pp. 966–978, 2009.

[100] L. Cucu-Grosjean, L. Santinelli, M. Houston, C. Lo, T. Vardanega, L. Kosmidis, J. Abella, E. Mezzetti, E. Quiñones, and F. J. Cazorla, "Measurement-based probabilistic timing analysis for multi-path programs", *Proceedings - Euromicro Conference on Real-Time Systems*, pp. 91–101, 2012.

[101] F. Wartel, L. Kosmidis, A. Gogonel, A. Baldovin, Z. Stephenson, B. Triquet, E. Qui, C. Lo, E. Mezzetti, I. Broster, J. Abella, L. Cucu-grosjean, T. Vardanega, F. J. Cazorla, and Quiñ.

[102] J. A. Stankovic, M. Spuri, K. Ramamritham, and G. C. Buttazzo, *Deadline scheduling for real-time systems: EDF and related algorithms*, 11-12. Springer Science & Business Media, 2012, vol. 460, p. 288.

[103] Y.-W. Seo, J. Kim, and R. Rajkumar, "Predicting dynamic computational workload of a self-driving car", in *Systems, Man and Cybernetics (SMC), 2014 IEEE International Conference on*, IEEE, 2014, pp. 3030–3035.

[104] S. L. Tan and B. A. Tran Nguyen, "Survey and performance evaluation of real-time operating systems (RTOS) for small microcontrollers", *IEEE Micro*, pp. 1–14, 2009.

[105] G. J. Holzmann, "Mars Code", *Communications of the ACM*, vol. 57, pp. 64–73, 2014.

[106] T. N. B. Anh and S.-L. Tan, "Real-Time Operating Systems for Small Microcontrollers", *IEEE Micro*, vol. 29, pp. 30–45, 2009.

[107] J. F. Ruiz, J. a. de la Puente, J. Zamorano, J. González-Barahona, and R. Fernández-Marina, "Open Ravenscar Real-Time Kernel", Universidad Politécnica De Madrid, University Of York, Construcciones Aeronáuticas, S.A., Tech. Rep., 2001.

[108] M. H. Neishaburi, M. Daneshtalab, M. R. Kakoee, and S. Safari, "Improving robustness of Real-Time operating systems (RTOS) services related to soft-errors", *2007 IEEE/ACS International Conference on Computer Systems and Applications, AICCSA 2007*, pp. 528–534, 2007.

[109] D. Silva, L. Bolzani, and F. Vargas, "An intellectual property core to detect task schedulling-related faults in RTOS-based embedded systems", *Proceedings of the 2011 IEEE 17th International On-Line Testing Symposium, IOLTS 2011*, pp. 19–24, 2011.

[110] Z. Zhengmao, Z. Yun, C. Ming, and S. Lei, "A Workload Model Based Approach to Evaluate the Robustness of Real-time Operating System", in *2013 IEEE International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing*, 2013, pp. 2027–2033.

[111] D. Abbott, *Linux for embedded and real-time applications*, Third. Oxford: Newnes, 2011.

[112] S. Rostedt and D. V. Hart, "Internals of the RT Patch", in *Proceedings of the Linux symposium*, vol. 2, Citeseer, 2007, pp. 161–172.

[113] B. O'Connor, "NASA software safety guidebook", *NASA Technical Standard NASA-GB-8719.13*, 2004.

[114] K. Baynes, C. Collins, E. Fiterman, B. Ganesh, P. Kohout, C. Smit, T. Zhang, and B. Jacob, "The Performance and Energy Consumption of Embedded Real-Time Operating Systems", *IEEE Trans. Comput.*, vol. 52, no. 11, pp. 1454–1469, 2003.

[115] Mendeley. (2015). Mendeley Desktop, [Online]. Available: `https://www.mendeley.com/`.

[116] F. Asplund, "Risks Related to the Use of Software Tools when Developing Cyber-Physical Systems", PhD thesis, KTH Royal Institute of Technology, 2014.

[117] M. Barabanov, "A Linux-based real-time operating system", New Mexico Institute of Mining and Technology, 1997.

[118] R. S. Pinto, F. J. Monaco, J. C. Faracco, and J. R. B. a. Monteiro, "Embedded Linux in real-time applications: Performance enhancements of experimental fully-preemptible capabilities over the standard kernel in a critical mobile system", *Proceedings - 2012 2nd Brazilian Conference on Critical Embedded Systems, CBSEC 2012*, pp. 76–81, 2012.

[119] R. Aitken, G. Fey, Z. T. Kalbarczyk, F. Reichenbach, and M. S. Reorda, "Reliability Analysis Reloaded: How Will We Survive?", in *Proceedings of the Conference on Design, Automation and Test in Europe*, EDA Consortium, 2013, pp. 358–367.

[120] T. Yoshimura, H. Yamada, and K. Kono, "Using Fault Injection to Analyze the Scope of Error Propagation in Linux", *IPSJ Online Transactions*, vol. 6, no. 2, pp. 55–64, 2013.

[121] E. V. D. Kouwe, C. Giuffriday, R. Ghituletez, and A. S. Tanenbaum, "A Methodology to Efficiently Compare Operating System Stability", *2015 IEEE 16th International Symposium on High Assurance Systems Engineering*, pp. 93–100, 2015.

[122] MINIX. (). MINIX, [Online]. Available: `http://www.minix3.org/`.

[123] Y. Lee, M. E. Jang, J. Kim, and Y. W. Park, "Performance analysis of software architectures with real-time kernel patches for the Rescue Robot", in *Control, Automation and Systems (ICCAS), 2014 14th International Conference on*, IEEE, 2014, pp. 460–465.

[124] F Proctor, "Introduction to Linux for Real-Time Control", Tech. Rep., 2002, pp. 1–78.

[125] H. Fayyad-Kazan, L. Perneel, and M. Timmerman, "Linux PREEMPT-RT vs. commercial RTOSs: how big is the performance gap?", *Journal on Computing (JoC)*, vol. 3, no. 1, 2014.

[126] RTAI. (). RTAI, [Online]. Available: `https://www.rtai.org/`.

[127] Xenomai. (). Xenomai, [Online]. Available: `http://xenomai.org/`.

[128] P. Hambarde, R. Varma, and S. Jha, "The survey of real time operating system: RTOS", in *Proceedings - International Conference on Electronic Systems, Signal Processing, and Computing Technologies, ICESC 2014*, 2014, pp. 34–39.

[129] A Barbalacel, A Luchetta, G Manduchi, M Moro, A Soppelsa, C Taliercio, A. Barbalace, A Luchetta, G Manduchi, M Moro, A Soppelsa, and C Taliercio, "Performance comparison of VxWorks, Linux, RTAI and Xenomai in a hard real-time application", in *Real-Time Conference, 2007 15th IEEE-NPSS*, IEEE, 2007, pp. 1–5.

[130] A. B. Atitallah, M. Kthiri, P. Kadionik, B Le Gal, H Lévi, and A Ben Atitallah, "Performances analysis and evaluation of Xenomai with a H. 264/AVC decoder", in *Microelectronics (ICM), 2011 International Conference on*, IEEE, 2011, pp. 1–4.

[131] S. Oikawa and R. Rajkumar, “Linux/RK: A Portable Resource Kernel in Linux”, *Proceedings of the{IEEE} Real-Time Systems Symposium*, pp. 111–120, 1998.

[132] H. K. Kim, R. Rajkumar, A. Colin, and G. Bhatia. (). Linux/RK, [Online]. Available: `http://rtml.ece.cmu.edu/redmine/projects/RK`.

[133] C. Garre, D. Mundo, M. Gubitosa, A. Toso, U. Calabria, M. Gubitosa, and A. Toso, “Performance comparison of real-time and general-purpose operating systems in parallel physical simulation with high computational cost”, SAE Technical Paper, Tech. Rep., 2014.

[134] W. Torfs and C. Blondia, “TDMA on commercial of-the-shelf hardware: Fact and fiction revealed”, *AEU - International Journal of Electronics and Communications*, vol. 69, no. 5, pp. 800–813, 2015.

[135] C. Garre, D. Mundo, M. Gubitosa, and A. Toso, “Real-Time and Real-Fast Performance of General-Purpose and Real-Time Operating Systems in Multithreaded Physical Simulation of Complex Mechanical Systems”, *Mathematical Problems in Engineering*, vol. 2014, no. 1, 2014.

[136] L. Deng, X. Zhao, C. Qi, and F. Gao, “A real-time walking robot control system based on Linux RTAI”, *International Conference on Advanced Mechatronic Systems, ICAMechS*, pp. 530–534, 2013.

[137] J. H. Koh and B. W. Choi, “On Benchmarking the Predictability of Real-Time Mechanisms in User and Kernel Spaces for Real-Time”, in *Computer Applications for Security, Control and System Engineering*, Springer Berlin Heidelberg, 2012, pp. 205–212.

[138] G. Rai and S. Kumar, “A Comparative Study : RTOS and Its Application”, *International Journal of Computer Trends and Technology (IJCTT)*, vol. 20, no. 1, pp. 41–44, 2015.

[139] N. Litayem and S. Ben Saoud, “Impact of the Linux Real-time Enhancements on the System Performances for Multi-core Intel Architectures”, *International Journal of Computer Applications*, vol. 17, no. 3, pp. 17–23, 2011.

[140] N. Asadi, “Enhancing the Monitoring of Real-Time Performance in Linux”, Mälardalen University, 2014.

[141] D. D. Niz, B. Andersson, and L. Wrage, “Cots multicore processors in avionics systems : challenges and solutions”, in *AIAA Infotech @ Aerospace*, Kissimmee, Florida, 2015.

[142] D. Faggioli, M. Trimarchi, and F. Checconi, “An implementation of the earliest deadline first algorithm in Linux”, *Proceedings of the 2009 ACM symposium on Applied Computing - SAC ’09*, p. 1984, 2009.

[143] Michael Kerrisk. (2015). Linux Programmer’s Manual, [Online]. Available: `http://man7.org/linux/man-pages/man7/sched.7.html` (visited on Jul. 31, 2015).

[144] M. S. Mollison, J. P. Erickson, J. H. Anderson, S. K. Baruah, and J. a. Scoredos, “Mixed-criticality real-time scheduling for multicore systems”, *10th IEEE International Conference on Computer and Information Technology*, pp. 1864–1871, 2010.

[145] R. Rybaniec and P. Z. Wieczorek, “Measuring and minimizing interrupt latency in Linux-based embedded systems”, in *Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments 2012*, vol. 8454, International Society for Optics and Photonics, 2012, 84540Y–84540Y.

[146] B. Blackham, Y. Shi, S. Chattopadhyay, A. Roychoudhury, and G. Heiser, “Timing analysis of a protected operating system kernel”, *Proceedings - Real-Time Systems Symposium*, pp. 339–348, 2011.

[147] A. Thekkilakattil, R. Dobrin, and S. Punnekkat, “The limited-preemptive feasibility of real-time tasks on uniprocessors”, *Real-Time Systems*, 2015.

[148] R. W. Butler and G. B. Finelli, “Infeasibility of quantifying the reliability of life-critical real-time software”, *IEEE Transactions on Software Engineering*, vol. 19, no. 1, pp. 3–12, 1993.

[149] AdaCore. (). Libraries and bindings, [Online]. Available: `http://www.adacore.com/gnatpro/toolsuite/libraries/` (visited on Aug. 25, 2015).

[150] J. Brown and B. Martin, “How fast is fast enough? Choosing between Xenomai and Linux for real-time applications”, *12th Real-Time Linux Workshop*, pp. 1–17, 2010.

[151] M. Piątek, “Real-Time Application Interface and Xenomai modified GNU/Linux real-time operating systems dedicated to control”, *Computer Methods and Systems*, 2007.

[152] H. Fayyad-Kazan, L. Perneel, and M. Timmerman, “Linux preempt-rt v2. 6.33 versus v3. 6.6: better or worse for real-time applications?”, *ACM SIGBED Review*, vol. 11, no. 1, pp. 26–31, 2014.

[153] OSADL. (). Open Source Automation Development Lab, [Online]. Available: `https://www.osadl.org/` (visited on Nov. 15, 2015).

[154] J. Altenberg, “Using the Realtime Preemption Patch on ARM CPUs”, in *11th Real-Time Linux Workshop (RTLW’09)*, 2009, pp. 229–236.

[155] Omron. (). Industrial automation, [Online]. Available: https://industrial.omron.eu/en/products/catalogue/automation{_}systems/machine{_}automation{_}controllers/sysmac-controllers/default.html (visited on Aug. 20, 2015).

[156] P. Regnier, G. Lima, and L. Barreto, “Evaluation of interrupt handling timeliness in real-time Linux operating systems”, *ACM SIGOPS Operating Systems Review*, vol. 42, no. 6, p. 52, 2008.

[157] L. Pike, S. Niller, and N. Wegmann, “Runtime verification for ultra-critical systems”, *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 7186 LNCS, pp. 310–324, 2012.

[158] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser, “Comprehensive formal verification of an OS microkernel”, *ACM Transactions on Computer Systems*, vol. 32, no. 1, pp. 1–70, 2014.

[159] SeL4. (). seL4, [Online]. Available: https://sel4.systems/Info/FAQ/{\#}14v (visited on Nov. 4, 2015).

[160] R. B. K. Dewar, “Safety-critical design techniques for secure and reliable systems”, in *Embedded Systems Conference*, San Jose, 2014.

[161] H. Zeng, M. Di Natale, P. Giusto, and A. Sangiovanni-Vincentelli, “Using statistical methods to compute the probability distribution of message response time in controller area network”, *IEEE Transactions on Industrial Informatics*, vol. 6, no. 4, pp. 678–691, 2010.

[162] B. L. Silva. (). Bbb-preempt-rt-kernel-patch, [Online]. Available: https://github.com/brnluiz/bbb-preempt-rt-kernel-patch (visited on Sep. 6, 2015).

[163] User:Tglx. (). Cyclictest, [Online]. Available: https://rt.wiki.kernel.org/index.php/Cyclictest (visited on Sep. 6, 2015).

[164] C. Williams and D. Sommerseth. (). Hackbench, [Online]. Available: http://man.cx/hackbench{\%}288{\%}29 (visited on Sep. 6, 2015).

[165] A. Kopytov. (). Akopytov/sysbench, [Online]. Available: https://github.com/akopytov/sysbench (visited on Sep. 6, 2015).

[166] J. Sparks. (). UDP Unicorn, [Online]. Available: http://sourceforge.net/projects/udpunicorn/ (visited on Sep. 6, 2015).

[167] W. a. Halang and J. Zalewski, “Programming languages for use in safety-related applications”, *Annual Reviews in Control*, vol. 27 I, pp. 39–45, 2003.

[168] A. Roßkopf and T. Tempelmeier, “Aspects of flight control software – a software engineering point of view”, *Control Engineering Practice*, vol. 8, pp. 675–680, 2000.

[169] H. Ciocarlie and L. Simon, “Definition of a High Level Language for Real-Time Distributed Systems Programming”, *EUROCON 2007 - The International Conference on Computer as a Tool""*, pp. 828–834, 2007.

[170] M. J. Pont, “An object-oriented approach to software development in C for small embedded systems”, *Transactions of the Institute of Measurement and Control*, vol. 25, no. 3, pp. 217–238, 2003.

[171] B. Ray, D. Posnett, V. Filkov, and P. T. Devanbu, “A large scale study of programming languages and code quality in github”, in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ACM, 2014, pp. 155–165.

[172] J. D. Orozco and R. M. Santos, *Real-Time Operating Systems and Programming Languages for Embedded Systems*. INTECH Open Access Publisher, 2012.

[173] P. Anderson, M. McDougall, and M. Zarins, “The use and implementation of coding standards for high-confidence embedded systems”, *IEEE Aerospace Conference Proceedings*, 2008.

[174] J. Barnes, *High Integrity Software: The SPARK Approach to Safety and Security*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[175] T. Erkkinen, “Developing high-integrity software in c and ada”, SAE Technical Paper, Tech. Rep. 724, 1999.

[176] D. J. Allerton, “High integrity real-time software”, *Proceedings of the Institution of Mechanical Engineers, Part G: Journal of Aerospace Engineering*, vol. 221, pp. 145–161, 2007.

[177] L. Hatton, “Language subsetting in an industrial context: A comparison of MISRA C 1998 and MISRA C 2004”, *Information and Software Technology*, vol. 49, pp. 475–482, 2007.

[178] W. J. Cullyer, S. J. Goodenough, and B. A. Wichmann, “The choice of computer languages for use in safety-critical systems”, *Software Engineering Journal*, vol. 6, no. 2, pp. 51–58, 1991.

[179] D. M. Jones, *The New C Standard: A Cultural and Economic Commentary*. Addison Wesley Longman Publishing Co., Inc., 2009.

[180] The Standish Group, "The Chaos Report", The Standish Group West Yarmouth, MA, Tech. Rep., 1995.

[181] M. Jørgensen and K. Moløkken-Østvold, "How large are software cost overruns? A review of the 1994 CHAOS report", *Information and Software Technology*, vol. 48, no. 4, pp. 297–301, 2006.

[182] R. L. Glass, "The Standish report: does it really describe a software crisis?", *Communications of the ACM*, vol. 49, no. 8, pp. 15–16, 2006.

[183] J. L. Eveleens and C. Verhoef, "The rise and fall of the Chaos report figures", *IEEE Software*, vol. 27, no. 1, pp. 30–36, 2010.

[184] J. Smith, "What About Ada ? The State of the Technology in 2003", Software Engineering Institute Carnegie Mellon University, Tech. Rep., 2003.

[185] J. Abraham, J. Chapple, and C. Preve, "Improving quality of Ada software with range analysis", *ACM SIGAda Ada Letters*, vol. 31, p. 69, 2011.

[186] B. M. Brosgol and A. J. Wellings, "A comparison of Ada and real-time Java for safety-critical applications", *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 4006 LNCS, pp. 13–26, 2006.

[187] J. F. Ruiz, "Ada 2005 for mission-critical systems", *Adacore report*, 2006.

[188] B. Brosgol and J. Ruiz. (2007). Ada enhances embedded-systems development, [Online]. Available: `http://www.eetimes.com/document.asp?doc{\_}id=1271495`.

[189] C. Cicalese, R. Weatherly, J. Sherrill, R. Bolling, K. Forbes, R. Grabowski, K. Ring, and D. Seidel, "A distributed, multi-language architecture for large unmanned ground vehicles", *ACM SIGAda Ada Letters*, vol. 28, p. 133, 2008.

[190] A. Srivastava, F. Woodard, and J. O'Leary, "Ada's Vital Role in New US Air Traffic Control Systems", *Imecs 2009: International Multi-Conference of Engineers and Computer Scientists, Vols I and Ii*, vol. I, pp. 1095–1099, 2009.

[191] P. Parkinson and F. Gasperoni, "High-Integrity Systems Development for Integrated Modular Avionics Using VxWorks and GNAT", in *Reliable software technologiesADA-Europe 2002: 7th Ada-Europe International Conference on Reliable Software Technologies, Vienna, Austria, June 17-21, 2002: Proceedings*, 2002, p. 163.

[192] B. M. Brosgol, "Ada in the 21st century", *CROSSTALK The Journal of Defense Software Engineering*, no. March, 2001.

[193] A. T. Chamillard and W. C. Hobart Jr, "Transitioning to Ada in an introductory course for non-majors", in *Proceedings of the conference on TRI-Ada'97*, ACM, 1997, pp. 37–40.

[194] M. P. K. Lawlis, "Ada and software maintenance", in *Software Maintenance, 1988., Proceedings of the Conference on*, IEEE, 1988, pp. 152–158.

[195] ISO/IEC TR 24772:2013, "Guidance to avoiding vulnerabilities in programming languages through language selection and use", ISO/IEC, Tech. Rep., 2013.

[196] ISO/IEC TR 15942, "Guide for the use of ada programming language in high integrity systems", ISO/IEC, Tech. Rep., 2000.

[197] N. Storey, "Design for Safety", English, in *Towards System Safety SE - 1*, F. Redmill and T. Anderson, Eds., Springer London, 1999, pp. 1–25.

[198] R. Chapman, "Correctness by construction: A manifesto for high integrity software", *Conferences in Research and Practice in Information Technology Series*, vol. 55, pp. 43–46, 2005.

[199] P Charland, D Dessureault, G Dussault, M Lizotte, F Michaud, D Ouellet, and M Salois, "Getting Smarter at Managing Avionic Software: The Results of a Two-Day Requirements Elicitation Workshop With DTAES", Tech. Rep. October, 2007.

[200] I. Gilchrist, "Attitudes to Ada – A Market Survey", in *ACM SIGAda Ada Letters*, vol. 19, ACM, 1999, pp. 229–242.

[201] L. Hatton, "Safer language subsets: An overview and a case history, MISRA C", *Information and Software Technology*, vol. 46, pp. 465–472, 2004.

[202] W. F. Tichy, P. Lukowicz, L. Prechelt, and E. a. Heinz, "Experimental evaluation in computer science: A quantitative study", *Journal of Systems and Software*, vol. 28, no. 1, pp. 9–18, 1995.

[203] J. Wainer, C. G. Novoa Barsottini, D. Lacerda, and L. R. Magalhães de Marco, "Empirical evaluation in Computer Science research published by ACM", *Information and Software Technology*, vol. 51, no. 6, pp. 1081–1085, 2009.

[204] W. F. Tichy, "Should computer scientists experiment more?", *Computer*, vol. 31, no. 5, pp. 32–40, 1998.

[205] A. Stefik, S. Hanenberg, M. McKenney, A. Andrews, S. K. Yellanki, and S. Siebert, "What is the foundation of evidence of human factors decisions in language design? an empirical study on programming language workshops", *Proceedings of the 22nd International Conference on Program Comprehension - ICPC 2014*, pp. 223–231, 2014.

[206] M. Tedre, “Computing as a science: A survey of competing viewpoints”, *Minds and Machines*, vol. 21, no. 3, pp. 361–387, 2011.

Appendices

Appendix A: Configuration and setup of the system.
Appendix B: Scripts for automated testing of the system.
Appendix C: Views about programming languages found in the literature.
Appendix D: Discussion about the use of evidence in computer science.

A Setting up the System

Installing cyclictest and hackbench

```
git clone git://git.kernel.org/pub/scm/linux/kernel/git/clrkwllms/rt-tests.git
cd rt-tests
make all
cp ./cyclictest /usr/bin/
cp ./hackbench /usr/bin/
```

Installing sysbench

```
apt-get install sysbench
```

Setting CPU governor

```
cpufreq-set -g performance
```

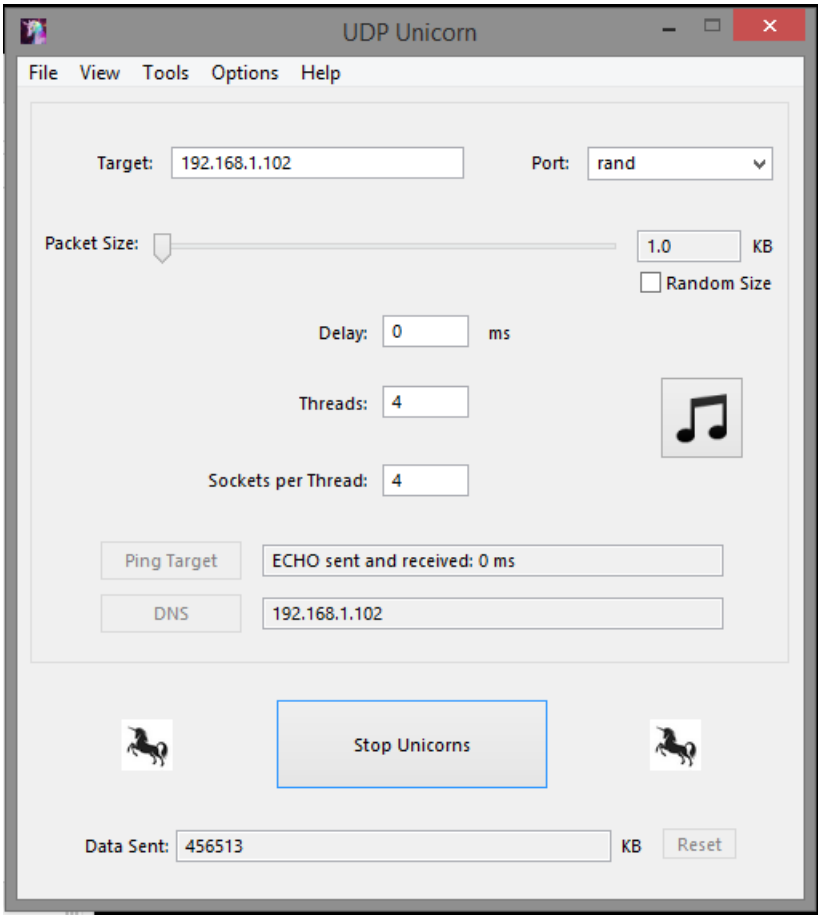

Figure 15: Settings used for stressing the BBB via Ethernet.

B Scripts for Testing

run_all_sysbench.sh

```
#!/bin/bash
echo "Starting stressed sysbench"
./sysbench_cyclic.sh
echo "Starting unstressed sysbench"
./sysbench_noload.sh
```

run_cyclic_except_ping.sh

```
#!/bin/bash
echo "Starting CPU"
./cyclic_cpu.sh
echo "Starting FileIO"
./cyclic_fileio.sh
echo "Starting Hackbench"
./cyclic_hackbench.sh
echo "Starting Memory"
./cyclic_memory.sh
echo "Starting NoLoad"
./cyclic_noload.sh
echo "Starting Threads"
./cyclic_threads.sh
```

cyclic_ping.sh

```
#!/bin/bash

dat=$(date +"%Y-%m-%d %H:%M:%S,%3N")
filedate=$(date +"%Y_%m_%d_%H_%M_%S_%3N")
cyclicpath=results/cyclic_ping/cyclic_ping_output_"$filedate"

echo "#START: $dat" > $cyclicpath
command="cyclictest -l3000000 -m -n -a0 -t1 -p99 -i200 -h1000 -q"
echo "#Command: $command" >> $cyclicpath

$command >> $cyclicpath

dat=$(date +"%Y-%m-%d %H:%M:%S,%3N")
echo "#FINISH: $dat" >> $cyclicpath
echo DONE
```

cyclic_threads.sh

```
#!/bin/bash
dat=$(date +"%Y-%m-%d %H:%M:%S,%3N")
filedate=$(date +"%Y_%m_%d_%H_%M_%S_%3N")
threadspath=results/cyclic_sysbench/threads_cyclic_output_"$filedate"
cyclicpath=results/cyclic_sysbench/cyclic_threads_output_"$filedate"

echo "#START: $dat" > $threadspath
echo "#START: $dat" > $cyclicpath

command_threads="sysbench --test=threads --num-threads=100 --thread-yields=1000000 run"
command_cyclic="cyclictest -l3000000 -m -n -a0 -t1 -p99 -i200 -h1000 -q"

echo "#Command: $command_threads" >> $threadspath
```

```
echo "#Command: $command_cyclic" >> $cyclicpath


$command_threads >> $threadspath &
threads=$!
$command_cyclic >> $cyclicpath
kill $threads

dat=$(date +"%Y-%m-%d %H:%M:%S,%3N")
echo "#FINISH: $dat" >> $threadspath
echo "#FINISH: $dat" >> $cyclicpath
echo DONE
```

cyclic_noload.sh

```
#!/bin/bash
dat=$(date +"%Y-%m-%d %H:%M:%S,%3N")
filedate=$(date +"%Y_%m_%d_%H_%M_%S_%3N")
cyclicpath=results/cyclic/cyclic_output_"$filedate"

echo "#START: $dat" > $cyclicpath
command="cyclictest -l3000000 -m -n -a0 -t1 -p99 -i200 -h1000 -q"
echo "#Command: $command" >> $cyclicpath

$command >> $cyclicpath

dat=$(date +"%Y-%m-%d %H:%M:%S,%3N")
echo "#FINISH: $dat" >> $cyclicpath
echo DONE
```

sysbench_cyclic.sh

```
#!/bin/bash
dat=$(date +"%Y-%m-%d %H:%M:%S,%3N")
filedate=$(date +"%Y_%m_%d_%H_%M_%S_%3N")


memorypath=results/sysbench_cyclic/memory_cyclic_output_"$filedate"
fileiopath=results/sysbench_cyclic/fileio_cyclic_output_"$filedate"
cpupath=results/sysbench_cyclic/cpu_cyclic_output_"$filedate"
threadspath=results/sysbench_cyclic/threads_cyclic_output_"$filedate"

cyclicpath=results/sysbench_cyclic/sysbench_cyclic_output_"$filedate"

echo "#START: $dat" > $memorypath
echo "#START: $dat" > $fileiopath
echo "#START: $dat" > $cpupath
echo "#START: $dat" > $threadspath

echo "#START: $dat" > $cyclicpath


command_memory="sysbench --test=memory --memory-block-size=1K --memory-total-size=4095M run"
command_fileio="sysbench --test=fileio --file-total-size=4G
--file-test-mode=rndrw --init-rng=on --max-requests=0 --max-time=600s run"
command_cpu="sysbench --test=cpu --cpu-max-prime=15000 run"
command_thread="sysbench --test=threads --num-threads=100 --max-time=600s run"

command_cyclic="cyclictest -m -n -a0 -t1 -p99 -i200 -h1000 -q"

echo "#Command: $command_memory" >> $memorypath
```

```
echo "#Command: $command_fileio" >> $fileiopath
echo "#Command: $command_cpu" >> $cpupath
echo "#Command: $command_thread" >> $threadspath

echo "#Command: $command_cyclic" >> $cyclicpath


$command_cyclic >> $cyclicpath &
cyclic=$!


function memstress {
for i in {1..10}
do
        $command_memory >> $memorypath
done
}

memstress
$command_fileio >> $fileiopath
$command_cpu >> $cpupath
$command_thread >> $threadspath
kill $cyclic


dat=$(date +"%Y-%m-%d %H:%M:%S,%3N")

echo "#FINISH: $dat" >> $memorypath
echo "#FINISH: $dat" >> $fileiopath
echo "#FINISH: $dat" >> $cpupath
echo "#FINISH: $dat" >> $threadspath

echo "#FINISH: $dat" >> $cyclicpath
echo DONE
```

sysbench_noload.sh

```
#!/bin/bash
dat=$(date +"%Y-%m-%d %H:%M:%S,%3N")
filedate=$(date +"%Y_%m_%d_%H_%M_%S_%3N")


memorypath=results/sysbench/memory_output_"$filedate"
fileiopath=results/sysbench/fileio_output_"$filedate"
cpupath=results/sysbench/cpu_output_"$filedate"
threadspath=results/sysbench/threads_output_"$filedate"


echo "#START: $dat" > $memorypath
echo "#START: $dat" > $fileiopath
echo "#START: $dat" > $cpupath
echo "#START: $dat" > $threadspath



command_memory="sysbench --test=memory --memory-block-size=1K --memory-total-size=4095M run"
command_fileio="sysbench --test=fileio --file-total-size=4G
--file-test-mode=rndrw --init-rng=on --max-requests=0 --max-time=600s run"
command_cpu="sysbench --test=cpu --cpu-max-prime=15000 run"
command_thread="sysbench --test=threads --num-threads=100 --max-time=600s run"
```

```
echo "#Command: $command_memory" >> $memorypath
echo "#Command: $command_fileio" >> $fileiopath
echo "#Command: $command_cpu" >> $cpupath
echo "#Command: $command_thread" >> $threadspath

function memstress {
for i in {1..10}
do
        $command_memory >> $memorypath
done
}

memstress
$command_fileio >> $fileiopath
$command_cpu >> $cpupath
$command_thread >> $threadspath
kill $cyclic


dat=$(date +"%Y-%m-%d %H:%M:%S,%3N")

echo "#FINISH: $dat" >> $memorypath
echo "#FINISH: $dat" >> $fileiopath
echo "#FINISH: $dat" >> $cpupath
echo "#FINISH: $dat" >> $threadspath

echo DONE
```

cyclic_memory.sh

```
#!/bin/bash
dat=$(date +"%Y-%m-%d %H:%M:%S,%3N")
filedate=$(date +"%Y_%m_%d_%H_%M_%S_%3N")
memorypath=results/cyclic_sysbench/memory_cyclic_output_"$filedate"
cyclicpath=results/cyclic_sysbench/cyclic_memory_output_"$filedate"

echo "#START: $dat" > $memorypath
echo "#START: $dat" > $cyclicpath


command_memory="sysbench --test=memory --memory-block-size=1K
--memory-total-size=4095M run"#ca 30 sec per test
command_cyclic="cyclictest -l3000000 -m -n -a0 -t1 -p99 -i200 -h1000 -q"

echo "#Command: $command_memory" >> $memorypath
echo "#Command: $command_cyclic" >> $cyclicpath

function memstress {
for i in {1..1000}
do
        $command_memory
done
}

memstress >> $memorypath &
memory=$!

$command_cyclic >> $cyclicpath
kill $memory

dat=$(date +"%Y-%m-%d %H:%M:%S,%3N")
```

```
echo "#FINISH: $dat" >> $memorypath
echo "#FINISH: $dat" >> $cyclicpath
echo DONE
```

cyclic_hackbench.sh

```
#!/bin/bash
dat=$(date +"%Y-%m-%d %H:%M:%S,%3N")
filedate=$(date +"%Y_%m_%d_%H_%M_%S_%3N")
hackpath=results/cyclic_hackbench/hackbench_output_"$filedate"
cyclicpath=results/cyclic_hackbench/cyclic_output_"$filedate"

echo "#START: $dat" > $hackpath
echo "#START: $dat" > $cyclicpath

command_hack="hackbench -l100000000"
command_cyclic="cyclictest -l70000 -m -n -a0 -t1 -p99 -i200 -h1000 -q"

echo "#Command: $command_hack" >> $hackpath
echo "#Command: $command_cyclic" >> $cyclicpath


$command_hack >> $hackpath &
hack=$!
$command_cyclic >> $cyclicpath
kill $hack

dat=$(date +"%Y-%m-%d %H:%M:%S,%3N")
echo "#FINISH: $dat" >> $hackpath
echo "#FINISH: $dat" >> $cyclicpath
echo DONE
```

cyclic_fileio.sh

```
#!/bin/bash
#Before starting test: sysbench --test=fileio --file-total-size=4G prepare

dat=$(date +"%Y-%m-%d %H:%M:%S,%3N")
filedate=$(date +"%Y_%m_%d_%H_%M_%S_%3N")
fileiopath=results/cyclic_sysbench/fileio_cyclic_output_"$filedate"
cyclicpath=results/cyclic_sysbench/cyclic_fileio_output_"$filedate"

echo "#START: $dat" > $fileiopath
echo "#START: $dat" > $cyclicpath

command_fileio="sysbench --test=fileio --file-total-size=4G
--file-test-mode=rndrw --init-rng=on --max-requests=0 run"
command_cyclic="cyclictest -l3000000 -m -n -a0 -t1 -p99 -i200 -h1000 -q"


echo "#Command: $command_fileio" >> $fileiopath
echo "#Command: $command_cyclic" >> $cyclicpath


$command_fileio >> $fileiopath &
fileio=$!
$command_cyclic >> $cyclicpath
kill $fileio

dat=$(date +"%Y-%m-%d %H:%M:%S,%3N")
echo "#FINISH: $dat" >> $fileiopath
```

```
echo "#FINISH: $dat" >> $cyclicpath
echo DONE
```

cyclic_cpu.sh

```
#!/bin/bash
dat=$(date +"%Y-%m-%d %H:%M:%S,%3N")
filedate=$(date +"%Y_%m_%d_%H_%M_%S_%3N")
cpupath=results/cyclic_sysbench/cpu_cyclic_output_"$filedate"
cyclicpath=results/cyclic_sysbench/cyclic_cpu_output_"$filedate"

echo "#START: $dat" > $cpupath
echo "#START: $dat" > $cyclicpath

command_cpu="sysbench --test=cpu --cpu-max-prime=10000000 run"
command_cyclic="cyclictest -l3000000 -m -n -a0 -t1 -p99 -i200 -h1000 -q"

echo "#Command: $command_cpu" >> $cpupath
echo "#Command: $command_cyclic" >> $cyclicpath


$command_cpu >> $cpupath &
cpu=$!
$command_cyclic >> $cyclicpath
kill $cpu

dat=$(date +"%Y-%m-%d %H:%M:%S,%3N")
echo "#FINISH: $dat" >> $cpupath
echo "#FINISH: $dat" >> $cyclicpath
echo DONE
```

C Programming Languages

This appendix provides the data that was used for evaluating Ada. Programming languages can sometimes be used for many domains, and sometimes only for very specific purposes. The focus here are on languages that are candidates for any systems that are either safety-critical, real-time, high-integrity or embedded. There are quite a few articles discussed here, this amount was deemed necessary because of lack of empirical evidence in computer science, see Appendix D.

The NASA guidebook on software safety [113] list beneficial features of Ada, which they say is one of the most commonly used languages in military and safety-critical applications. Some of these features are:

- multiple styles possible like object-oriented programming and functional (structural) programming techniques.
- Compiler validation.
- Mixed language support.
- Language restriction ability and more.

When it comes to C they say it is extremely popular because of the flexibility and support environment, the non-technical factors such as availability of tools, vendors and experienced programmers are good as well. The C definition is however considered to not be at the level necessary for safety-critical applications. There exists many dialects of C, causing integration problems between code reuse on different platforms. Despite it's drawbacks it has been used successfully in safety-critical applications, but the effort on the developer is larger for verifying the software. They mention that C++ suffers from many of the same drawbacks as C. Java has limitations like its virtual machine, garbage collections and it is not standardized by major standard groups. Their conclusion is that Ada95 (and Ada83) with the subset SPARK is the safest languages. However , since it's not the most popular it can be difficult to find and keep good programmers, therefor other languages are often chosen. Their advise is to be aware of the limitations of other languages if Ada is not used, extra time needs then to be spent on inspection, analysis, testing and the use of coding standards.

A popular book [30] on real-time systems and languages focuses on C, Java and Ada with the motivation that these language are being discussed because they are actually used.

In [38] is a literature review conducted on programming languages. They recognize that developers use different languages such as Assembly, Ada, C and C++.

Their conclusions is that however that C is the most dominant and de facto language for developing resource-contrained embedded systems.

Halang et al [167] mentions that Ada is often used on a industrial scale in safety-related system and mostly military applications. Also C is one of the most commonly used languages, but it has weaknesses such as:

- very weak typing,
- severe problems with dynamic memory allocation and pointer arithmetic
- logical expressions are defined, but a logical data type is not,
- potential ambiguities in if conditions,
- increment/decrement operators, etc.

To overcome the problem of trustworthiness of software they propose a dedicated subset of PEARL 90.

Rogers [14] compares Ada and C++ for safety critical applications. The conclusions is that Ada is designed for those type of systems and C++ is not. Early data indicates that coding in C++ is always more difficult and expensive compared to the initial plans for a project.

> “In the world of safety-critical programming the languages supported by the least expensive labor force have, for many years, been C and C++. There is a glut of C and C++ programmers in the world of software development. They pour out of colleges, universities, and technical schools like a mighty river. Unfortunately, use of the C and C++ languages results in significant hidden costs and safety risks due to the clumsiness and lack of precision the languages provide as safety-critical programming tools.
>
> *James S. Rogers [14]* ”

Brosgol [45] discusses the use of Ada for high-integrity applications and concludes that all the Ada revisions at that time (83,95 and 2005) meet the requirements of those systems.

Naiditch [37] compares Ada83, Ada95, C, C++ and Java against a set of factors, Ada95 scores high on all the factors which none of the other languages does. Further a lists of 11 misconceptions about programming language is discussed, some of them are:

- Misconception 2: C is the most efficient language since it is closer to assembly code

- Misconception 6: Java is a great language because the same code can run on windows 95, Solaris, Unix, Macintosh, or just about any platform
- Misconception 7: Using operating system (os) services for concurrent programming is just as good (if not better than) using special language constructs
- Misconception 9: Java is small, simple, and easy-to-learn
- Misconception 11: In terms of life cycle costs, it does not matter what programming language is chosen

According to [39] C is the primary language at many organizations, NASA/JPL included, therefor a list 10 programming rules are described targeted for C. Note that the NASA guidebook considered Ada as the safest language, but even in NASA C is the primary language, at least at JPL division.

In [1] it is said that 80% of all companies in the embedded-software domain are using C, over 40% are using assembler and to some degree C++ is used. Java is said to be increasing its usage for GUI applications. A rule of thumb is presented that can estimate the amount of defects in software that needs to be found and removed: Raising the amount of functions points to the power of 1.22. If one wants the number of defects per lines of code (LOC or KiloLOC) one must consider how many statements there are per function point, and the amount of statements per line of code, which is language specific. For C they say that the KLOC/Function point ratio is 150 to 1 (depending on the dialect), which evaluates to 30 defects per KLOC for C.

This rule of thumb indicates that the amount of defects in an application only depends on the number of function points, not the language. The authors does not however explicitly mention which languages this is applicable for, other than C. But it gives cause to be careful when comparing defects/KLOC between languages since in theory a language that needs more KLOC for an implementation will show less defects/KLOC, this is shown for a real use case between Ada and C in [32].

For flight control software [168] discusses the use of C, C++, Java and ADA. The conclusion is that there are many small problems in inappropriate languages that in end translates to difficult problems. However, using Ada reduces this aspects to becoming trivial as a whole to the designer, allowing for more focus on application problems instead of bug fixing.

Ciocarlie et al. [169] lists a set of factors for real-time languages and compares major languages to these features. They say that only a small numbers of sys-

tems are developed in Ada, which they say is unfortunate because Ada would simplify embedded development compared to C++. However, Ada is having a mini-comeback, particularly because of open-source compilers for Linux (a view is shared by [17]). Java is said to be receiving some attention because its attractive features, however in its original form it's unsuitable for real-time application because its terrible timing behavior due to the garbage collector. The Real-Time specification for Java might be predictable enough to introduce Java as real contender, however they say it depends on how much extra complexity and reduced features it realizes.

Pont [170] considers resource-constrained embedded systems, it is recognized that O-O languages has attractive features, but the languages are typically to resource consuming. An object oriented approach for C is presented, and the conclusion is that C is the most appropriate language. A recent study [171] however shows that C, C++ and especially Objective C is more failure prone than other languages.

The most popular programming languages for multi-core embedded systems are discussed in [35] which they considered to be Assembly, C, C++, Ada, Java and Python. They acknowledge that the C11 and C11++ standard adds multithreading to the language, which previously was provided only by external libraries. C is the language of choice for many embedded systems due to it's ability to directly access hardware. C++, Ada, Java and Python have features that give rise to increased code size and performance degradation, making them unsuitable for resource-constrained systems. They however say that some features can be disabled to increase performance, for example the run-time checks in Ada. Python is not suitable for time-critical applications, and it also requires a lot of memory making it unsuitable for small embedded systems, but it's used by embedded system developers to create prototype-code quickly, and for high-end embedded systems like smartphones. They say Java is a costly language that is unsuitable for system that are either real-time, safety-critical or has limited resources. It's like Python popular for smartphones and web applications. For real-time safety-critical applications Ada should be considered.

Kornecki et al. [42] discusses programming languages for software that needs to be certified against some standard, often safety-critical. For those applications the contenders are Ada, C, C++ and Java, for which DO-178B certifiably has been claimed. They say Ada seems be the most advanced in that respect with its history of certification and validation from the start in the eighties.

Brosgol [46] derives general language requirements from the DO-178B standard,

in other words safety-critical and high-security software. Ada, C, C++, and Java is then evaluated against these requirements. The focus of the analysis is how these languages can be subsetted to ease the certification of applications. He says MISRA-C has been successfully used in safety-critical systems, but C has significant drawbacks that are intrinsic and that cannot be removed by subsetting. C++ is considered to be in many ways better than C, two subsets are common, JSF C++ and MISRA C++, however the same drawbacks for MISRA-C can be found in those subsets. They say Ada was designed for safety-critical systems which avoids many drawbacks found in C and C++, it is continuing to be widely used in military and commercial avionics.

His opinion is that the full Ada language is however too large to be used for these type of systems. According to him this can be solved by using Ada's unique way of excluding features, no extra tool is required, this is supported by any standard compiler. A subset can be defined in any way the application requires, additionally an official certifiable subset profile (Ravenscar) is included in the standard. The disadvantages of Ada are said to be mainly non-technical; its usage is smaller than the other languages and the tool vendor community is smaller. SPARK(an Ada subset) is the language that best meets the requirements, however it suffers from the same non-technical disadvantages like Ada, also it lacks expressibility. Java has some benefits like less implementation dependencies, but they conclude that compared to the other languages it has the most significant challenges for safety-critical applications.

Orozco et al. [172] discusses real-time languages and the support for them in RTOSes. The languages considered are: Ada, C, C++, RT Java and in very specific applications Assembler. When using Java, both the JVM and JIT are unsuitable because of of unpredictable behaviour and its memory foot print. The more promising options is the use of Java co-processor or processor. The co-processor translated Java code in to native code to the coupled CPU. The Java processor executes Java bytecode directly as native code. This can solve memory and performance issues, however it requires the system to use specific hardware. Ada is considered a good choice for embedded systems with safety-critical or real-time applications, but mainly for systems with large resources in terms of processing power and memory, for small embedded systems its unsuitable. It's used in aerospace, defense, medical, rail-road and nuclear systems.

They say C++ is a very powerful language, but it's difficult to manage all its features. There is a lack of compilers compared to C. In the end developers prefers using C in spite of its limitations compared to C++. Of the five languages, the trend is reducing the contenders to C, C++ and RT Java. Finally, they conclude that the benefits of C, like the availability on almost every hardware platform,

close interaction with the hardware and its libraries makes is an excellent choice for developing real-time embedded applications.

Anderson et al. [173] discusses software from a safety perspective, enormous effort is being spent on validating software, but still almost all software stills contains defects. For safety-critical software the typical ways for development can't be used. The use of coding standards is discussed to see how they can be used to improve the software development. For these systems C and C++ are the dominating language, even though the are unsuitable:

> “ Despite the inherent lack of safety in the definition and implementation of the C and C++ programming languages, they remain the dominant languages used in most embedded systems programming today. Languages such as Ada offer much better guarantees, and correctness-by-construction methods such as SPARK [174] have proven effective at increasing code safety, but economic forces and institutional momentum have prevented widespread adoption of these approaches. C and C++ are expected to remain the language of choice for the foreseeable future.
>
> Anderson et al. [173]. ”

Erkkinen [175] compares Ada and C with its subsets(SPARK and MISRA), the view is that Ada, and languages like Modula, are more suited for high-integrity software but that C can used in these applications as well with proper development.

In [160] the view is that C, C++, Java and Ada are all unsuitable for safety-critical software. The languages are too complex to be used for these applications, but by using subsets they can be reduced to a level where the languages are fully understood. The problem with this approach could however lead to a language that is to restrictive to be used for the intended application [176]. There is balance between functionality and the ability to verify the software.

Many of the papers above suggests using subsets and coding standards to increase reliability, but this might have the adverse effect than intended. To see if the code adheres to a standard tools can be used that checks the code, but for example the popular MISRA standard have a very high false-positive ratio(mandatory to non-mandatory rules), for every genuine fault detected there are 50 false-positive faults in the MISRA C 1998 subset according to Hatton [177]. Correcting these false-positives can lead to re-injection of faults.

In the end, according to Hatton, it is not unlikely that more faults are introduced by fully adhering to the MISRA C 1998 compared to not using it at all. Hatton says that one of the goals of MISRA C 2004 was to reduce this effect, however as he reports the real to false-positive ratio has actually increased, making it even worse than MISRA C 1998. If the newest version, MISRA C 2012 has made any progress in this area remains to be seen. Still, coding standards are widely adopted, as an example the NASA Mars rover project employed six levels of coding standards according to [105]. For the highest level, the code had to comply with all the MISRA rules. In the code review process approximately 10,000 peer comments and 30,000 tool generated reports where discussed, of these 84% led to changes in the code.

Clearly, defects will be present (and found) even if the project adheres to coding standards. But perhaps the most important aspect about coding standards is not to try reducing the amount of defects made by the programmer, but instead restricting the language to make it possible to use tools to analyze the code efficiently. In [39] is the first rule is to forbid recursions, this will allow code analyzers to find the limits of stack use. Some organizations like NASA can afford to have the code reviewed by both humans and tools, but this is not the case for many other organizations/companies where the tool solution is the only viable option. Making it possible to use those tool is therefor key, coding standards helps in doing that.

According to Hatton [177], one problem with C and C++ is that the compilers successfully compiles faulty code that should be rejected, which they say has led to discussion about if these language should be used at all in safety-related system. However it is noted that using compilers for quality measure is not sufficient since the fact is that some of the world's most reliable systems, Linux, has been written in C.

A paper from 1991 [178] discusses the choice of programming languages for safety-critical systems. A set of technical factors are defined, and then each language is evaluated against those factors. It is advised to not use C, its definition is too undefined and vary between applications which makes it significantly worse than assembler. A subset for C++ could be a contender, but no conclusions could be made at that stage. Ada is considered a good choice, however unrestricted use cannot be recommended, instead a subset should be used. The most appropriate languages, with subsets, are in descending order: ISO Pascal, ADA, Modula-2, CORAL66. Of these languages, only Ada is still being considered by more recent papers cited in this thesis.

Another paper [62] once again evaluates Ada, C and C++ against a set of factors, this time for the use in weapons systems. Ada scores well on the technical factors of the languages. But the lack of tools, libraries, compilers, education and training makes it an unattractive language compared to C/C++. It is estimated that the cost for developing a compiler for a new target platform would be 1 million dollars and adding to that, 18 months to field it. The conclusions at that time, late 90's, was that Ada's positions had been weakened compared to other languages, and the trend would continue in that direction.

In [36] a comparison is made between different languages. For the overall rating Ada scores the highest, followed by C++,Java and then C. C scores particular low for reliability and safety. Further, Ada provides the best mixed language support. Ada, C and C++ are all standardized language supported by standards organization, however it is noted that there are important difference between the standards. For Ada the standard was written before the language was developed, for C and C++ the standard was adopted to the language afterwards. This results in many dialects for C/C++ with significant differences between them, which is said to be a possible source of problems with for example portability.

Jones [179] argues that one aspect of costs for Ada have been overlooked, namely if considering all projects, successful and canceled, Ada is not the most cost effective option. The argument is that Ada's savings are made on the maintenance part of the projects, so for the canceled projects Ada's cost will be higher. If one can determine the amount of canceled and successful projects its possible to calculate the total cost, Jones gets this information from the well known Chaos Report [180], however the validity of that report have discussed in [181–183], the conclusion is that those figures can't be trusted.

Smith [184] recognizes Ada as respected language in safety-critical software development. It has the potential to lower life cycle cost, but in reality its more likely too increase cost due to lack of compilers, tools and experienced programmers. The use of Ada is said to be decreasing and it has almost totally disappeared as a teaching language in computer science.

Ada has been commended for it's real-time capabilities, however [79] points out that Ada has a lack of functionality. The delay statement in Ada for the timing of tasks is only a lower bound of the scheduling, there is no support for guaranteeing the upper bound. A task can therefore miss it's deadlines. However, it is now possible to monitor the timing properties of a task, like start time and executing time. A handler can then be used that activates if a deadline is missed [26, 30, 45].

Ada is according to [185] a key component in large complex projects like the Boeing 787, F-22 Raptor, F-35 JSF, all projects with several million lines of codes. These systems have strong requirements on testing and verification, the authors give concrete examples of how tools for Ada can be used to verify and test the software, in particular range analysis is dicussed.

Brosgol et al. [186] compares Ada and Real-Time Java. Neither full Ada or Java is suitable for safety-critical systems, this the opinion of [187] as well. Ada can be subsetted by using pragmas that restricts features, however the flexibility of this can cause certification problems when using certified code in other implementations. Still, they say Ada is expected to be a language of choice when developing safety-critical applications, and because of the broad use of Java in many systems this could lead to a mixed language solution using both Ada and Java for developing safety-critical systems. This suits well with the requirements of mixed language systems, Ada has great support for using code written in other languages [188]. A real mixed language application is shown in [189] where C, Ada and Java are all used for developing the software for unmanned ground vehicles. More about how to interface Ada with other languages is described in the reference manual [28], the Annex B shows both general interface support and how to use packages with specific support for C, C++, COBOL and FORTRAN.

The core of the FAA's air traffic control was written in Ada in unprecedented short time within budget [190]. Its success in aviation is supported by [191], they say Ada's abstraction from processor and systems architectures has helped the avionics programmes to produce portable code. Also, the Ada95 standard goes further than the C/C++ counterparts in trying to achieve compiler compatibility, however closer examination of the compiler still needs to be done because of potential vendor lock-in solutions.

There are worries about start up costs when switching to a new language, the start up cost will be less when the programmers are skilled in the chosen language, however any skilled programmer in C or C++ can come up to speed to Ada fairly quickly by tutorials, book, courses. A five day course can make a professional programmer proficient in Ada [192]. Indeed, when the USAF academy was transitioning to Ada from Pascal in programming courses there was a concern about faculty inexperience using Ada. However it turned out at Ada was sufficiently similar to Pascal and other imperative languages that enabled the faculty to quickly develop the skill needed for teaching the courses [193].

Different languages are popular in different domains, according to [3] are small embedded applications often programmed in C and C++. For larger real-time

systems Ada dominates. In Germany Pearl is common for industrial automation. Ada's approach of having real-time support built in the language rater than an API is said to be beneficial for static analysis by the compiler leading to higher quality of the code and cost-effective production. They say Ada remains the language of choice for many high-integrity applications but it has never regained its momentum it had in the early years, there are less programmers around and it's not taught as much in universities.

In an review of Ada from 1988 the conclusions was that Ada was not the perfect language, but it was the best currently available [194]. It's not perfect today either, the ISO/IEC technical report [195] on guidelines for avoiding vulnerabilities shows that all the languages they considered has vulnerabilities, there are however many vulnerabilities in C that is for example not present in Ada due to the different languages designs. Clearly, no perfect language exits, but some has more vulnerabilities than others.

Ada is considered to be unique in it's compliance to the attributes needed for a language to be used for high-integrity software [196, 197]. Ada is said to be standing on its own as a language that provides the framework needed for static analysis and correctness by construction [196]. Indeed, in [198] it is reported that the defect rate for projects using the correctness-by-construction (CbyC) approach are extremely low, this while maintaining a productivity higher than the industry standard. However it is noted that Ada is not suitable for this, instead SPARK should be used. Ada 2012 does however introduce the concept of contract programming with its pre and post conditions [27] making it more suitable for the CbyC approach.

In [29] it is mentioned that Ada is the only language in common use where the real-time constructs plays a main role of the language, other languages like C/C++ relies on external libraries and Java has only weak support. They say Ada is the language of choice for many complex systems, such as the Boeing 787.

Sward [59] discusses the rise, fall and persitance of Ada. He says that the language has declined in popularity since its glory days in the mid 90's, the lack of effective compilers is mentioned as one of the reasons, it has however persisted in the domain of safety-critical and high integrity software: *It is acknowledge that Ada has not become an widely accepted mainstream developing language, however it has persisted in the domain of safety-critical, high-integrity and large scale complex system.*

From a workshop about avionic software [199] the conclusions about usage of

programming languages was that Ada is still used but not taught at many places anymore. Assembly is present as well. C is loosing ground to C++, and the use of Java is virtually non existent.

A market study[200] evaluates of the use of Ada. There is almost no criticism about the language itself, but it's clear that the factors outside the language has significant weaknesses such as lack of tools, trained staff and libraries. C and C++ on the other hand are the most popular choice, even though it receives criticism for being unreliable languages. It is said that strength of having a wide range of cheap tools, libraries and trained staff is so important that it seems to outweigh the technical weaknesses.

D Evidence in Computer Science

In philosophy, religion and science there are many questions that never will be answered to the satisfaction of all people. In computer science there is one that's been of much debate, namely what programming language to use. Programming languages pops up like weeds in a garden, often being very similar to each other. Old features, libraries and algorithms that already exists gets implemented in new languages. The view of [66] is that the duplication effort is enormous, the peer-review process to ensure originality of scholarly papers seems to be non-existent for this field. C has been an influential language in the development of other languages like C++, Javascript and Java. Unfortunately all these languages have reproduced the defects of C that was originality reported in 1982 by one of the architects, Dennis Ritchie [201].

> “ Indeed, one of my major complaints about the computer field is that whereas Newton could say, "If I have seen a little farther than others, it is because I have stood on the shoulders of giants," I am forced to say, "Today we stand on each other's feet." Perhaps the central problem we face in all of computer science is how we are to get to the situation where we build on top of the work of others rather than redoing so much of it in a trivially different way. Science is supposed to be cumulative, not almost endless duplication of the same kind of things.
>
> *Richard Hamming 1968 Turning Award Lecture* ”

It is not difficult to find so called 'flame wars' on the subject, people arguing why 'their' programming language is better, without providing evidence. This is perhaps what you can expect from discussions at forums and blogs. However, in the scientific community this is often the case as well, which is surprising, it is for example not guaranteed that a language under the scrutiny of a standard organization is being enhanced by the means of science; Stefik et al. [66] argues that the C++ ISO committee is being pseudo-scientific.

Its not easy to make an sound decision on what language to use based on facts when there is a lack of evidence. In [202] 400 research papers, about computer science, are analyzed regarding the use of experimental evaluation. Their findings where disappointing: Over 40% of articles about new designs and models completely lacked such experimentation and the fraction was even higher, 50%, for the samples related to software engineering. This study was done in 1995,

two decades ago, but has things changed since then? A similar study was conducted by [203] where papers from 2005 where evaluated in the same way as was done in 1995, the conclusion was that not much had changed, computer science still lacked empirical evaluation.

> “There are plenty of computer science theories that haven't been tested. For instance, functional programming, object-oriented programming, and formal methods are all thought to improve programmer productivity, program quality, or both. It is surprising that none of these obviously important claims have ever been tested systematically, even though they are all 30 years old and a lot of effort has gone into developing programming languages and formal techniques.
>
> *Tichy 1998 [204]* ”

A recent study [205] evaluates papers from 1986 to 2012 on the use of evidence for human factors in language design. Their findings are similar to ones above, worse actually: The analysis of papers from 2009 to 2012 shows that approximately only 28.1% of them that has empirical information.

Why is this the case? Perhaps it partly could be because computer science has had an identity crisis, starting from its birth, contributing to many different views of what methodology to use [206]. In [66] the academic institutions are mentioned as not providing the incentive for evidence based research:

> "Further, given that academic institutions often rely considerably on paper counts, literally the number of papers an academic publishes as a metric, academics are directly discouraged from working to alleviate hard and long-standing problems that might take years of hard study if it leads to only one, or a few, publications. In other words, while we admittedly have no direct observations, we wonder whether modern methods for evaluating tenure and promotion are having a negative impact on the quality of work in the academic literature".
>
> Stefik and Hanenberg [66]

The take away from this is that extra attention is required when studying the literature related to computer science and not jumping to conclusions too fast when reading about promising findings in research papers.