Performance and Robustness Engineering:

From A Conflict Towards Fair Coexistence

Rainer Gerlich, Ralf Gerlich

BSSE System and Software Engineering, Auf dem Ruhbuehl 181, 88090 Immenstaad, Germany, Tel. +49 (7545) 91.12.58, Fax: +49 (7545) 91.12.40, E-Mail: gerlich@t-online.de, URL: http://home.t-online.de/home/gerlich

Abstract. Performance engineering aims to meet a system's resource constraints which implies to make the code as lean as possible. Robustness engineering has to support fault identification and fault recovery. To make a system sufficiently robust, code needs to be added. Hence, robustness engineering is in conflict with a system's performance. This paper will discuss this potential conflict in the context of operating systems. Operating systems are significantly impacting performance and robustness properties according to what they are providing or do not support. Current practices and related impacts are identified and recommendations are given how this conflict can be minimised.

Keywords: Performance engineering, robustness engineering, operating systems, shared address space, separate address space, pre-emptive scheduling, cooperative scheduling

Preface

This paper represents a revised version of a contribution to the "2. Workshop on Performance Engineering" [PE2001]. It considers new experience (dated August 2001) in this subject since the submission of the paper by January 2001, and discussions we had, especially during the workshop.

We started witth the sub-title "A Potential Conflict?" and added a question mark at the end. When finalising the paper we deleted the question mark, because then it was clear for us it is a conflict. At the time of the presentation we stated it is not a conflict, because both topics have to be considered well if we want to get reliable and performant software. This is our opinion now.

By the discussions we had and the experience we made it is our beliefe that robustness issues are still underestimated and we would like to take this opportunity to point out the importance of robustness enginering. We have experienced that multiple effort is required when software is used which does not check for valid conditions - either due to dormant bugs and fault propagation which could not be identified due to missing checks - or due to user errors which remain undetected.

We hope that by this paper we can convince software engineers of the importance of these topics and can give a guideline how both aspects can be sufficiently considered.

Finally, we would like to describe the current situation according to what we have seen so far by a rather provocative statement which however bears a lot of truth in it: it seems that craftsmen are better trained for quality issues and they are getting more pressure to care about it than software engineers during their education. And we do hope that this will change in future.

An explanation for this situation is: nearly no tools exist which make (poor) quality visible. Hence, nobody can really conclude on the quality of software when it is delivered. However, we suppose that lack of such tools is related to lack of user demand. Consequently, this causes a deadlock. We did a first step to escape from this situation by an approach we presented on the first performance workshop in 2000 [PE2000]

1 Introduction

Performance engineering is essential for successful completion of a software project [Scholz/Schmietendorf 2000]. To meet a customer’s functional and performance requirements is a pre-condition for a system’s final acceptance. To succeed at the end evaluation of performance is continuoulsly needed during development. This issue guides an engineer to continuously care about performance matters.

A system’s robustness can only be evaluated by fault injection during the development phase[1] or by presence of faults during the operational phase. While poor performance is already recognised when the system is in use under real conditions, poor robustness is not of such evidence. As far as no fault occurs robustness cannot be measured. Consequently, robustness can only be demonstrated if a lot more is done than usually is required for normal operations. In fact, due to the increased effort robustness engineering is limited today to high integrity systems as needed in the domain of aerospace, transportation and nuclear applications.

Currently, the importance of performance engineering is more deeply recognized, after it was underestimated for a long time in the past ([Scholz/Schmietendorf 2000] and the references listed there). While for performance engineering the situation is now continuously improving, it is not for robustness engineering. Higher priority is still assigned to performance than to robustness issues in case of non-high-integrity systems, presumably because the impacts are less evident under normal operational conditions. Faults only occur sporadically, therefore their impacts remain unknown if the system is not stimulated by fault injection or stress testing.

But even if such testing techniques are applied a certain risk remains that a fault is not detected. Then it is important to limit its impact when it occurs, e.g. by prevention or limitation of fault propagation.

Performance and robustness issues may significantly be constrained or enhanced by the capabilities of an operating system (OS). We will discuss below that it is of extreme importance for the overall properties of a software system which capabilities an OS supports and how it tries to solve the conflict between performance and robustness. For instance an OS can optimise its **own** peformance to the disadvantage of the overall system performance or robustness.

Chapter 2 classifies operating systems according to their performance and robustness properties and describes the overall impact.

[1] Bugs which are present during software testing and which can be identified are as helpul as intentionally injected faults

Chapter 3 gives recommendations on how performance and robustness issues can be harmonised. Finally, chapter 4 makes conclusions on the previous discussion and the current status of performance and robustness engineering.

2. The Impact by Operating Systems

As the functions of an operating system are frequently called by all application software they need to perform well. This leads to an approach, e.g. in case of VxWorks [VxWorks] and Linux RT [Linux RT]), which does not prevent fault propagation due to use of a shared address space or omittance of error checks[2].

In order to decrease the criticality of fault propagation if an OS does not sufficiently support robustness issues, checks need to be added by the application which in fact make the overall performance worse than in the case when the OS provides such checks, apart from the fact that more development effort is needed.

The following Table 1 lists the essential characteristics of a number of OS regarding performance and robustness.

The following classification criteria are applied:

- memory organisation

  shared or separate address space
- scheduling

  cooperative or pre-emptive
- basic error checking

  check on invalid addresses, address verification

The next sections provide detailed information on the classification criteria and the observed consequences for the listed operating systems.

2.1 The Classification Criteria

The selected classification criteria are described by the following sections regarding their impact on performance and robustness.

2.1.1 Memory Organsiation

Two principal organisation schemes are analysed: shared and separate address space.

Regarding performance a shared address space minimises the data access time. Data can directly be accessed by each application, no messages need to be exchanged to access data of another application as it is required in case of a separate address space.

However, from a robustness point of view sharing of the same address space is an extreme disadvantage. A fault in an application can easily corrupt memory of another application or of

[2] This list is non-exhaustive. The list only includes such OS which have been used in practice for a number of projects. The fact that such problems occured for the listed OS neither does imply that such OS are worse than others nor that other OS not listed here are more robust or performant.

the kernel - which is the worst case, but happens frequently. In latter case the complete system may halt or crash.

It was observed that address violations inside an application, e.g. by invalid pointers or indices of arrays, will cause a system crash at a high probability in case of a shared address space. Unfortunately, such violations cannot be detected by C compilers at compile time. Moreover, due to the system crash no information is available indicating the reason of the crash. Therefore additional tests must be performed to get an idea on what caused the crash. This significantly increases the development effort.

Additional means for pre-mortem tracking must be added to get a chance for fault identification. If such a fault occurs only sporadically, there is no chance at all for identification. Then the engineer has to wait until the fault can be reproduced, he needs to identify the error condition which also requires a lot of effort, or he needs to find the bug by analysis.

Such sporadical faults are much more probable on systems with a shared address space because then all the memory can be accessed, irrespectively whether it is a code and data area or unused. In case of an address space limited to the code and data area of a program uninitialized or malformed pointers access critical locations or illegal addresses at a higher probability. Hence, the chance is higher to fix such bugs. If only a small portion of the available memory is used, then most address errors will lie outside the critical area, so that the chance to detect them is lower. Consequently, a low ratio between used and unused memory increases the probability of sporadic faults occuring after delivery of the software. Of course, the probability to fix them is increased if checks are added, and this is matter of robustness engineering.

In our projects the effort was increased by a factor of 2 .. 5. In fact, more time was needed to get software running in a shared address space which was already tested for a separate address space. E.g. when porting software from Un*x to VxWorks the effort for testing of the specific software related to VxWorks interfaces was much higher than the effort needed to test the complete software on Un*x.

To summarize: From a performance point of view a shared address space would be the best choice. Regarding robustness a separate address space would be preferable. The possible fault propagation in case of a shared address space has a dramatic negative impact on a system’s robustness and development effort. In worst case no recovery from a crash is possible by the system itself, while it is in case of a separate address space [3].

2.1.2 Scheduling

Cooperative and pre-emptive scheduling are the schemes most frequently supported.

Cooperative scheduling means that an application only gets control of the processor when the active application releases it. This makes response times indeterministic. In worst case, an application can block the whole system for ever. Then rebooting is the only possibility to recover from such a situation. From this point of view cooperative scheduling brings in negative aspects on performance and robustness. However, it also has some positive impacts on robustness: the scheduling of applications is much simpler and bears less risks.

[3] provided that the fault does not occur in the system’s kernel

Pre-emptive scheduling allows to react immediately on a certain event. A supervisor can always take the control on the system even if an application program hangs. These are the positive aspects regarding performance and robustness.

The higher complexity introduced by pre-emptive scheduling impacts the robustness from a principal point of view. More development and testing effort and more sophisticated algorithms are needed, especially in the area of prevention and resulution of conflicts resulting from several tasks accessing the same resource.

In case of cooperative scheduling no recovery from blocking is possible, while it is in case of pre-emptive scheduling [3].

2.1.3 Basic Error Checking / Address Verification

The meaning of “basic error checking” is limited here to “address verification”. All OS - as known so far - do not support more than such checks. The impact on performance is limited as the checks are mostly done by hardware which raises exceptions in case of a wrong address. Also, the checks are limited to whether the address is in the allowed address space or not. Hence, the checks do not imply detailed range checks. However, in case of uninitialised indices or pointers the resulting address is - at high probability - not part of the application’s address space and an exception is raised by which the fault can be identified. Such checks are only possible in case of a separate address space.

An easy way to check proper initialisation of a pointer would be to initialise it by 0 (NULL). Some OS or hardware check on NULL pointers, so that non-initialised pointers which are out-of-range of the valid address space can be identified this way. If there is no such support, then it is hard to detect such bugs[4].

Basic address verification has nearly no impact on performance because the checks are usually performed by hardware, while the robustness is increased due to the higher possibility for fault identification, especially during development. It is our experience that the probability to detect an adressing bug is higher the smaller the allowed address space of the application is.

Having tested on a VxWorks platform with 128 MB we ported our software to a 16 MB platform and immediately observed a crash due to an address error. The explanation is: in case of a smaller physical address space hardware may detect the illegal address, and the probability to corrupt a program's code or data is much smaller when a lot of address space is not used.

[4] Although tools exist which search for non-initialised pointers it is not always possible to identify such bugs by source code analysis. E.g. a number of irrelevant warnings may make the whole report unreadable so that it is difficult to find the serious messages.

2.1.4 Summary on Evaluation Criteria

Table 1 summarises the results of the previous discussion.

Feature	Type	P	R	Performance	Robustness
Address Space	shared	+	-	more performant data access	fault propagation
	non-shared	-	+	less performant data access	no fault propagation by data access violations
Scheduling	pre-emptive	+	-	faster reaction on events	higher complexity of code
	non-pre-emptive	-	-	potentially slower reaction on events	danger of blocking
Error Checking	implemented	-	+	additional checks take time	additional checks allow to detect, identify and remove faults
	not / poorly implemented	+	-	no overhead by checks	faults may not be detected and may propagate, high risk for dormant faults

Table 1: The Impact of the Classification Criteria on Performance (P) and Robustness (R)

2.2 Observations

A number of OS have been used or analysed in the past which represent a set of combinations of above classification criteria. The impacts on performance and robustness are discussed by the following sections. Table 2 summarises our observations.

2.2.1 VxWorks

VxWorks is a real-time operating system with the following characteristics: shared address space, pre-emptive scheduling, no (support of) address verification.

From a principal point of view the combination of “shared address space” and “pre-emptive scheduling” is the absolute worst case regarding robustness, especially from a developer’s point of view.

Moreover, no range checks are performed or supported, neither for the OS functions nor the user’s functions although they are urgently needed due to high negative impacts on robustness by the chosen memory organisation and scheduling mode.

In total, this leads to big instabilities during the development phase due to possible propagation of application errors into the OS kernel or inside the OS kernel in case of overflow of OS buffers. Nevertheless, it is possible to provide a robust and stable application on top of VxWorks, but to the expense of increased development effort and additional provisions made by the application to avoid fault propagation.

From a performance point of view VxWorks does not take the full advantage of pre-emption because most actions are related to the system's clock period (system tick) which is usually 1/60 s and asynchronous features are not fully supported like for asynchronous BSD sockets. In fact, time progresses only in units of the system tick unless a high-resolution timer is available and the user takes care about it himself.

Also, the advantage of a shared address space, the direct access of data, causes overhead on data access or the system's data organisation. As no direct comparison is availabe (implementation of the same application for shared and separate address space while the other conditions remain the same) a direct conclusion is not possible on whether a shared address space gives a real benefit for performance or not.

The overhead we see is related to organisation of data of process instances which share the same data names. Following solutions are used in practice to cope with that problem:

- arrays are needed to keep the data independent of each other

  but access of an array is less efficient than of a non-indexed data,

  the data structures become more complex, this increases development effort and potentially decreases robustness
- overloading of data names is needed

  which requires temporary storage of such data in the task frame of each process instance

  this causes an overhead because for each task (thread) switch, such data must be retrieved and restored
- due to pre-emption synchronisation is needed for data shared by tasks which probably requires execution of much more instructions than can be saved by direct data access.

In case of a separate address space such measures are not needed.

Also, in most cases the direct access of data as means of intra-processor[5] communication is not applied, because instead message exchange is used for reasons of process synchronisation.

However, when interfacing with I/O devices the shared address space is of advantage because no context switches are needed when an application needs to read data from or write to a peripheral device.

A positive feature of VxWorks compared to other OS like Un*x-based systems is its rather small kernel size, a criterion which is out of scope for this discussion, however.

Provided that bugs which may cause corruption of memory have been identified during development by additional measures of the project, and no overload situation occurs by which OS buffers can overflow, VxWorks will run rather stable.

An overhead of about 200 .. 500 % was observed compared to Un*x OS to get rid of the most dangerous bugs. It is doubtful if the shared address space really helps to increase performance due to above mentioned organisational overhead. On the other side, the architectural concept of the OS significantly compromises a system's robustness.

---

[5] "intra-processor" communication means local communication on the same processor, while "inter-processor" communication covers communication between processes residing on different processors

OS	Experience	Memory Organisation	Scheduling	Error Checking	Biggest Impact	Comments
VxWorks	practice	kernel and application share the same address space	priority-based preemptive	no boundary-checks, no check on valid addresses	crash of the complete system, potential corruption of file system and damage of peripherals	may lead to a highly instable system if no additional means are added by the application, development effort is rather high due to insufficient means for fault prevention and identification, no post-mortem dump possible a priori
Linux	practice	separate address space for kernel and application	preemptive (time slices)	address checks, in most cases identification of out-of-range addresses	only the faulty application crashes	kernel survives a crash of the application post-mortem dump possible system can fully recover from the fault no impact on other applications
Solaris	practice	same as for Linux	same as for Linux	same as for Linux	same as for Linux	same as for Linux
Linux RT	analysis	kernel and time-critical tasks share the same address space	priority-based preemptive	no checking of boundaries, protected null-page	same as for VxWorks, if a time-critical application fails	unknown but possibly similar to VxWorks
Win 3.11	practice	separate address space for kernel and application	cooperative	address checks, boundary checks	crash of the complete system due to potential corruption of other address space, blocking of system operations possible corruption of file system possible	no post-mortem dump possible, may lead to an highly instable system, memory protection may be overridden
Win 95 Win 98	practice analysis	same as for Win 3.11	preemptive	address checks, boundary checks	same as for Win 3.11	same as for Win 3.11 corruption of file system encountered in case of Win95, user data were lost, OS re-installation was required
Mac OS 9	practice	kernel and application share the same address space	cooperative	no check on valid addresses	same as for Win 3.11	high development overhead as for VxWorks corruption of file system encountered

Table 2: OS Characteristics

2.2.2 Linux

Linux is a Unix-based system supporting a separate address space, basic address checking and pre-emptive scheduling.

Pre-emption is mainly based on time slices, therefore its response times may be larger than for VxWorks which applies priority-based pre-emptive scheduling. However, handlers for asynchronous inputs are supported which allow a faster response time than in case of VxWorks which does not support this feature[6]. Time resolution is determined by the available timer, there is no need for the user to spend extra efforts on getting the highest possible time resolution.

The separate address space requires message exchange for intra-processor communication. But - as mentioned above - this makes no principal difference for an OS which is based on shared address space because for synchronisation purposes message exchange is needed (used) in any case. Shared memory can be introduced in Linux applications, but then synchronisation is required as for shared address space due to pre-emption.Access of I/O devices is managed by the kernel and context switches are required. This is a disadvantage compared with a shared address space.

However, the separate address space brings in a higher degree of robustness and suppresses fault propagation by corruption of data or code of foreign processes. Especially, in case of a fault in an application, the OS kernel and the other processes are not affected. Hence, the system can autonomously recover from such a fault, even from non-anticipated software faults[3]. This is a significant advantage.

2.2.3 Solaris

Solaris is a Unix-based system and all the considerations are valid as given for Linux above. In practice, no signficant differences were observed regarding the points of discussion.

2.2.4 Linux RT

Linux RT ([Linux RT]) (and other real-time derivatives of Linux) takes an ambivalent approach. As Unix-based system it supports a separate address space but it introduces a shared address space for access of peripherals. This way it inherits the advantages of direct access of peripherals regarding performance, but it also inherits the disadvantages regarding robustness.

Linux RT allows application code to share the same address space with the OS kernel while processes with separate address space are also supported.

If the application code which shares the kernel's address space is well tested (as good as the kernel itself) there should be no (big) impact on robustness. From this point of view the conceptual mix fairly considers the needs of performance and robustness. Consequently, the size of code sharing the address space of the kernel should be as small as possible, and all functionality not dealing directly with I/O access should be moved to processes executed in an address space separate from the kernel's one.

[6] This observation was made for version 5.3 of VxWorks

2.2.5 MS-Windows

The different versions of MS-Windows represent different conceptual approaches. Regarding scheduling all schemes are used starting with cooperative scheduling for Win 3.11, supporting pre-emptive scheduling for Windows NT and ending with a mixed approach with cooperative and pre-emptive scheduling for Windows 95/2000.

In case of Win3.11 the system can be blocked due to purely cooperative scheduling. In some cases it is possible to cancel a blocking task, in most cases it is not. Blocking also can occur if a modal input box of the user interface is hidden (for what ever reason) and there is no possibility to access the desktop to get it into foreground.

Applications have their own separate address space, but it is possible that application faults can propagate into the kernel from where they seriously can corrupt memory or the file system. For Windows 95 a number of corruptions of the hard disk were encountered which caused loss of user data and required re-installation of the OS.

2.2.6 Mac OS

Up to Mac OS 9 a shared address space and cooperative scheduling is supported. OS X will support a Unix-like concept but as it is not officially available, there is no experience so far.

Mac OS 9 gets all the disadvantages of a shared address space and cooperative scheduling, hence it is rather instable and may block. As for VxWorks the development effort is very seriously impacted and a significant cost and schedule overhead must be taken into account.

Due to use of cooperative scheduling no synchronisation of data access is needed, message exchange as known from VxWorks and Un*x is not supported. This leads to exchange of pointers instead of the data and increases the performance due to a very efficient data exchange/access mechanism. However, as Apple does not guarantee a shared address space in a mid-long-term perspective (possibly, Mac OS X will no longer support it), developers which want to remain compatible with future Mac OS versions will avoid direct data access or they have to isolate this access method. Hence, this potential benefit does not become a real benefit under maintenance considerations.

We have encountered a number of situations where use of an OS function introduced a system crash, and we had to spend e.g. effort of one week for several cases to identify it is not a fault in our application. There is a high risk, that use of an OS function may lead to a problem, possibly due to dormant bugs. Also, we had a number of problems due to misleading documentation, by which we gave wrong inputs and missing checks on such wrong inputs.

3. Classification of Concepts and Recommendations

As can be seen by the facts listed in chapter 2 performance and robustness are really in conflict with each other. Higher performance may compromise robustness and by adding means to achieve higher robustness performance is suffering. So the question is: What is the optimum conceptual mix which meets best the performance and robustness issues?

Robustness is an essential property which should not be compromised in any case. This is also true for performance, of course. The simplest solution to solve this conflict is to use more efficient hardware. However, this only can or will be a case-by-case solution, not a principal one.

For better understanding of the problem, section 3.1 will summarise and classify the observations and remarks of chapter 2. Section 3.2 gives general recommendations.

3.1 Classification of the Concepts

A classification of the concepts as implemented by the investigated OS is done according to the following criteria:

- no overall performance benefit

  such concepts can be excluded from the final discussion
- benefit either for performance or robustness

  such concepts bear a potential conflict regarding optimisation of performance and robustness

Unfortunately, a concept is still missing which can optimise both, performance and robustness, without any compromise.

3.1.1 Concepts With No Real Performance Benefit

This is a category of concepts which intend to increase performance but in fact end up with less performance compared with other approaches which look less performant at the first glance. Hence, such concepts are out of scope for performance tuning from a general point of view. This is even more essential if such concepts seriously compromise robustness.

3.1.1.1 Fully Shared Address Space and Pre-Emptive Scheduling

In case of a “fully shared address space” the direct data access is only of limited advantage. As was pointed out the shared address space requires data organisation schemes which add an overhead compared with a separate address space. Moreover, if combined with pre-emptive scheduling the required synchronisation mechanisms cause much more overhead than can be saved by direct access. Finally, according to our experience direct data access is not the most frequently used mechanism of data exchange. Message passing, like used in case of a separate address space dominates by far. Hence, no real performance benefit can be seen by this concept, while it significantly compromises robustness.

3.1.1.2 Omitted Checks

Omitting of checks may be considered as a means to achieve higher performance. However, this is not true.

E.g. the documentation of VxWorks frequently states that checks have been omitted in its kernel to increase the performance. The consequence is that a user has to add own checks to be able to identify bugs, because it is very difficult to find the reason of a bug if faults can propagate.

The missing checks, e.g. on valid parameters on OS functions, have to be added by the user. However, such add-ons seem to be less performant than in case the OS implements them - apart from the fact that the user needs to spent more effort. Consequently, the application suffers from the missing checks. The only - very questionable - benefit is that the OS itself shows good performance to the disadvantage of the overall application.

There is a special category of such missing checks which bear a high risk and require much effort to recover from. These are related to functionality excluded by an OS, or an application in general. In such cases the documentation states "a user MUST NOT do this". We found that such a warning is usually the only measure undertaken to prevent misuse of a certain feature. But in practice either this statement is hidden somewhere in the documentation, it is forgotten or it is not recognised at all that it is of relevance Then it is very difficult to identify the reasons for a crash and much effor tis needed. And at the end, the user is guilty becasue the application provider does refer to his warning or exclusion.

Although such a mix of "forbidding, but not checking" is already bad enough, we even have observed a higher level of criticality: to disable the functionality in certain cases and/or to require use of other functions, but not to document it.

In our opinion the correlation of limited functionality and missing documentation of this restriction is not surprising, both events have the same roots: the exceptional occurence of a restriction. The user falls into the trap because he cannot imagine at all that such an exception is needed or meanigful. The OS provider forgets to document it because it is also exceptional for him. If no checks prevent erroneous use of such software, the chance to introduce a bug is very high, and the effort to fix it is high, too.

3.1.2 Ambivalent Concepts

These concepts either optimise towards performance OR robustness, but do not satisfy both needs. The two principal categories “address space” and “scheduling” are considered independently.

3.1.2.1 Limited Shared Address Space

A “limited shared address space” as provided by Linux RT is an optimum solution regarding fast access of peripheral hardware and avoidance of fault propagation as far as no better concept is available which allows fast access for a separate address space. If the quality of the software which shares the kernel’s address space is high, it is the best solution which is currently possible.

3.1.2.2 Separate Address Space

A “separate address space” is the optimum solution if access of peripherals is not very time-critical. Due to separation of processes the robustness is high because propagation of faults into other processes is not possible (except for CPU overload).

3.1.2.3 Pre-Emptive Scheduling

Pre-emptive scheduling requires significant overhead for synchronisation, especially on data access. The alternative of message passing is also less performant. Due to pre-emption a number of provisions must be taken to avoid side-effects. This increases the complexity and may compromise robustness. However, pre-emption is the only way to escape from blocking and to ensure short response times. Therefore, also from a robustness point of view, pre-emptive scheduling is a “must”.

3.1.2.4 Cooperative Scheduling

In case of cooperative scheduling less complexity is required which increases robustness from a principal point of view. However, the response times become indeterministic and blocking is possible. This compromises both, performance and robustness.

3.2 Recommendations

As no concept fully satisfies the needs of both, performance and robustness, recommendations can only be given either towards performance or robustness.

3.2.1 Highest Priority To Robustness

In this case priority should be given to a combination of “separate address space” and “pre-emptive scheduling”. Then a maximum of protection against fault propagation and blocking is achieved, a system can survive even if one of its processes completely fails.

3.2.2 Highest Priority To Performance

When priority is given to performance a “limited shared address space” with “pre-emptive scheduling” is the best choice. This results in still sufficient robustness and ensures fast data access and response times.

If checks signficantly compromise performance they should be made removable (“compilable comments”). This allows to reliably detect bugs during development and decreases the probability of fault propagation during the later operational phase, although fault propagation is not fully excluded during operation.

This is a good solution for applications for which engineers claim "we will not get the needed performance if we add checks". To ommit checks would be the worst thing they could do. As user of such software we have observed that a number of bugs remain dormant if no checks are performed. Then such bugs come up after delivery and during operation by the user. Therefore it is important to add checks and to provide the right means to remove them if needed without introducing new faults at time of removal.

3.3 Evaluation Summary

Table 3 summarises the conclusions on OS properties regarding performance and robustness. In case of performance the response time is taken as evaluation criterion.

	Good	**Medium**	**Poor**
Performance	Linux RT, VxWorks	Un*x, MS-Windows	Mac OS 9
Robustness	Un*x	Linux RT, MS-Windows NT	VxWorks, Mac OS 9, MS-Windows 95

Table 3: Classification of the OS Regarding Performance and Robustness

4. Conclusions

Performance and robustness are in conflict to each other. A number of OS concepts have been analysed which confirms this conclusion. The term "conflict" is used here to describe the fact that an optimisation of performance, e.g. by lean code which ommits error checks, decreases robustness. Vice versa, robust code decreases performance.

However, from a quality point of view the term "conflict" should not be understood such that a decision towards one of both topics is possible, only, thereby neglecting the needs of the other one. It is a very challenging issue to cover both at the highest level possible. This requires a good understanding of the conflict potential. By this paper we have identified the impacts in the area of operating systems, and to provide feasible solutions for a fair coexistence of both topics.

The criteria for classification of the operating systems regarding performance and robustness were “organisation of the address space”, “scheduling”, “address verification”.

A “fully shared address space” has a lot of disadvantages and is considered as out of scope for performant and robust system implementations, while a “limited shared address space” seems - currently - to be the optimum solution regarding performance and robustness. “Pre-emptive scheduling” seems to be the most appropriate scheduling approach.

Checking mechanisms have been identified as a must for error identification and avoidance of fault propagation. If checking is not supported by the OS to increase performance, the checks which need to be added by the user may result in less overall system performance than in the case the OS does sufficient checking. But it is highly recommended to apply checking on valid conditions, and to add own checks if needed. If performance is signficantly compromised “removable checks” are recommended, i.e. checks which are only active during testing and integration.

In case of a “fully shared address space” the development effort is significantly higher than for a “separate address space”. If combined with “pre-emptive scheduling” this is absolutely the worst case regarding robustness and development effort.

We have observed a serious underestimation of robustness issues in the area of software engineering outside the domain of dependable applications. Tools are missing to evaluate the quality and robustness of software. Quality issues and standards are understood as recommendations which merely will be checked. Quality standards and guidelines are provided, but benchmarking on the benefit of such exercises are missing, hence corrections and improvements are not possible.

In fact, quality assurance is understood as a passive discipline, giving guidleines only and believing that this is sufficient to ensure quality of the deliverables. This understanding of quality assurance is also reflected by the current practices of education in the area of software engineering.

We do hope that by above considerations we can encourage engineers to do more to achieve higher quality of software and to prove this by concrete facts without dropping performance issues.

References

Linux: Information can be obtained from http://www.linux.org

Linux RT: Information can be obtained from http://www.rtlinux.org/˜rtlinux

Mac OS:Information about Mac OS 9 and OS X can be obtained from http://www.apple.com

PE2000: Workshop "Performance Engineering inder Softwareentwicklung", PE2000
May 15, 2000, DeTeCSM, Darmstadt, Germany

R.Gerlich: Built-In Performance and Robustness Engineering Capabilities by a Formalised and Automated Software Development Process
updated version:
Lecture Notes in Computer Science LNCS 2047, Springer Verlag, 2001,
A.Scholz, A. Schmietendorf (editors)
R.Gerlich: Performance and Robustness Engineering and the Role of Automated Software Development

PE2001: 2. Workshop Performance Engineering in der Softwareentwicklung, PE 2001
April 26, 2001, Universitaet der Bundeswehr Munich, Germany,
Rainer Gerlich, Ralf Gerlich: Performance and Robustness Engineering: A Potential Conflict

Sckolz, A. Schmietendorf, A.: Aspekte des Software Engineerings - Aufgaben und Inhalte. pp. 33-40 In: Rainer Dumke, Claus Rautenstrauch, Andreas Schmietendorf, André Scholz (Ed.), Tagungsband 1. Workshop Performance Engineering in der Softwareentwicklung (PE2000), May 15, 2000, Darmstadt, Germany

Solaris: SunSoft Inc. 2550 Garcia Avenue, Mountain View, CA 94043, USA

VWKS: TORNADO / VxWorks, WindRiver Systems, Inc. 1010 Atlantic Avenue, Alameda, CA 94501-1153, USA

Win 3.11, Win95, Win98: Information can obtained from http://www.microsoft.com