CONFIGURABLE HYBRIDKERNEL FOR EMBEDDED REAL-TIME SYSTEMS

Timo Kerstan, Simon Oberthür
Heinz Nixdorf Institute, University Paderborn
Fürstenallee 11, 33102 Paderborn, Germany
timo.kerstan@uni-paderborn.de, zottel@uni-paderborn.de

Abstract: When designing a kernel for an operating system the developer has to choose between an microkernel or monolithic kernel approach. Bases for the decision is mostly the tradeoff between security and performance. Depending on application demands and on the available hardware a microkernel or a monolithic kernel approach or something between is desired. In this paper we present a hybrid kernel for embedded real-time systems which can be configured to the application demands in an easy way. To realize the hybrid kernel we present a technique to guarantee memory access in *O(1)* with virtual memory. With our approach the same codebase can be used for system services to be placed either in userspace or in kernelspace.

Keywords: real-time operating system, microkernel, monolithic kernel, virtual memory, memory management unit, MMU, translation look-aside buffer, TLB

1. INTRODUCTION

The operating range of embedded systems is varying from simple toys up to aircrafts. Hence real-time, performance and security demands are varying from negligible to indispensable. On the other hand most security features implemented in modern real-time operating systems [5] are at the expenses of performance, if not implemented or supported in dedicated hardware. A usual approach to increase security is application/operating system separation in embedded systems by using a kernel architecture. To increase security of service of the operating system, the services can be separated from basic operating system functionality as well. This is implemented by placing each service in userspace in a own separated address space.

When using the kernel approach the question is which functionality to keep in the kernel and which functionality to put in the userspace. As motivated from the security point of view putting as much as possible functionality in separated userspace sections is desired. From the performance point of view

Kerstan, T., Oberthür, S, 2007, in IFIP International Federation for Information Processing, Volume 231, Embedded System Design: Topics, Techniques and Trends, eds. A. Rettberg, Zanella, M., Dömer, R., Gerstlauer, A., Rammig, F., (Boston: Springer), pp. 279–288.

putting as little functionality in separated userspace sections is desired, to avoid expensive context switches and communication between different address spaces. Which direction of this tradeoff to choice depends highly on the applications demands and scenario.

In this paper, we present an extension of our fine granular configurable RTOS DREAMS [2]. We introduce a flexible configurable kernel implementation: A configurable Hybrid Kernel. Our approach allows an efficient way to configure which service to place in kernel and which in userspace. The main goal is to give the developer the possibility to adapt the kernel to his special needs regarding fault isolation for security and speed for efficiency.

To make system services in userspace as fast as possible high-performance context switching is crucial. High-performance context switch requires hardware specific tweaks. This paper presents an approach for our operating system on the PowePC405 architecture.

We first present the related work (2) which inspired our approach. The offline configurabilty of our approch is realized by the Skeleton Customization Language (SCL) (3.1). After presenting our hybridkernel (3.2), we shortly discuss an approach to minimize context switching time on our hardware architecture and to guarantee memory access in *O(1)* using virtual memory (3.3).

2. RELATED WORK

In the field of operating systems the pros and cons of microkernel vs. monolithic kernel approaches have been largely discussed (eg. [1], [8]). Looking at the market leading operating systems and real-time operating systems today both kernel designs can be found. The benefit of the microkernel approach is the isolation of operating system components in userspace. In the microkernel only basic and fundamental functions are implemented other features are separated in different address spaces. This increases fault tolerance, because bugs can only affect a small area of other code. In a big monolithic kernel a bug can effect on the whole functionality in the huge kernel. A small microkernel minimizes the number of bugs in the critical kernel routines because lines of code and number of bugs is related to each other (cp. [7]). The design paradigm of micro kernels (component seperation/development and standardized interaction model) implies a more structured design. Of course such a design can likewise be used in a monolithic kernel. The most mentioned benefit of monolithic kernel is the performance, because in microkernel each context switch is an overhead. The performance of micro kernel implementations can be massively improved by optimizing the kernel to special hardware. A technique to minimize context switch time is shown in this paper in section 3.3 or from Hildebrand in QNX [3].

Kea

Kea [9, 10] was designed for experimentation with kernel structures. It allows a fine-grained decomposition of system services and a dynamic migration of services between different adress spaces (Domains in Kea) during runtime. This includes the migration of services between userspace and kernelspace to accelerate frequently used services. To provide transparency for the developer and the system inter-domain-calls (IDCs) are introduced. These IDCs make use of so called portals which map the function call to the appropriate domain and service providing this function. This is done in a dynamic manner as service migration is possible during runtime. For real-time operating system this dynamic behaviour implies not deterministic latencies of function calls. Nevertheless the basic idea of a hybrid kernel providing safety, modularity and performance depending on the system requirements is the same as stated in this paper. This paper additionally focuses on embedded and real-time constraints.

Emeralds

EMERALDS[1] [11, 12] was developed at the University of Michigan and is a scientific prototype achieving an efficient microkernel for embedded systems with low resources. The main goal is to provide extremely fast system calls which are comparable to regular function calls. This shall expunge the disadvantage of slow system calls in microkernel architectures compared to monolithic kernel architectures.

To accelerate the system calls, EMERALDS always maps the kernel into every user process. The section of the kernel is protected against misuse of the user process. The advantage of this mapping is that at every system call no switch of the address space is necessary anymore reducing the complexity of the context switch from a user adress space to the kernel adress space. Due to the fact that no exchange of the virtual mappings has to be done, every pointer keeps its validity preventing the relocation of pointers and the data transfer between userspace and kernelspace. The developers of EMERALDS showed that the resulting complexity of a system call in EMERALDS is in the same dimension as a regular function call.

We need to note that EMERALDS is not really a microkernel as most of the services are realized in the kernelspace. This leads to a lack of security as an erroneous service may cause the whole system to crash.

[1] Extensible Microkernel for Embedded ReAL-time Distributed Systems

3. A CONFIGURABLE HYBRIDKERNEL FOR EMBEDDED REAL-TIME SYSTEMS

The configurable hybridkernel presented is based on the operating system DREAMS which is written in C++. The kernel is build on the minimal component ZeroDREAMS, is offline configurable and uses virtual memory to separate processes into different address spaces. The offline configurability is based on our Skeleton Customization Language (SCL).

3.1 Skeleton Customization Language

SCL [2] is used for the configuration of our hybrid kernel architecture. The SCL is a framework for offline fine granular source code configurability and was introduced together with the development of DREAMS. The main aspects provided by SCL are the configurability of the superclass and the members of a C++ class. The framework uses a configuration file to generate code out of the configuration file and the assigned source files.

To configure the *superclass* of a *subclass* the following expression is used

```
SKELETON SubClass IS A [VIRTUAL] path/SuperClass[(args)];}
```

The keyword *VIRTUAL* is optional and instructs the SCL framework to generate virtual methods. It is possible to specify the constructor of the superclass to be used for initialization by providing the parameters of the corresponding constructor.

To add a configurable member object to a class the developer has to provide the following line for each configurable member object:

```
MemberName ::= [NONE|DYNAMIC] path/ClassName[(args)];
```

That allows to configure the class of the member object and the constructor to be used. One of the key features here are the optional keywords *NONE* and *DYNAMIC*. The keyword *NONE* allows to define the abscence of this member object and the keyword *DYNAMIC* states that the member object is initialized when the first access occurs.

Because of the possibility to declare members absent the developer needs to consider this when writing his code. Therefore SCL creates managing methods for every configurable Member:

```
hasMemberName();
getMemberName();
setMemberName();
```

A short example shows how the resulting code would look like after the SCL Framework processed it. The example is used in a Single-Threading Environment in which no scheduling is necessary. This abscence of the scheduler is

described by the keyword *NONE*. The used architecture accesses the memory through a MMU (Memory Management Unit) and therefore a member object for the configuration of this MMU is available.

```
SKELETON PowerPC405 IS A Processor{
    MMU ::= path/PowerPC405MMU;
    Scheduler ::= NONE path/Scheduler
}

class PowerPC405 : Processor{
    ...
    hasMMU(){return true;};
    getMMU(){return radio;};
    setMMU(PowerPC405MMU arg){mmu = arg;};
    ...
    hasScheduler(){return false;};
    getScheduler(){return NULL;};
    setScheduler(){};
    ...
};
```

3.2 Hybridkernel

Using SCL allows a fine granular configurability within the source code of our hybridkernel. This configurability is used in the so called *Syscall Dispatcher*, which is a demultiplexer for system calls. It delegates the system calls to a configured service object. The handlers are realized as configurable member objects of the *Syscall Dispatcher* and can easily be exchanged within the SCL configuration. This allows to create an offline configurable kernel.

Our hybridkernel is able to place the services of our OS inside the kernelspace or in the userspace depending on the developers needs for security and performance. Therefore we are using proxies of the services which encapsulate only the communication, provided by the mikrokernel architecture (e.g. message passing). These proxies delegate the system calls to the appropriate service skeleton within the userspace and need to ensure that the semantics of the system calls are realized correctly (e.g. blocking system calls). An example configuration for a service moved from kernelspace to userspace is depicted in Figure 1. It is important to state that the proxy and the skeleton can be generated using state of the art techniques as they are used i.e. in Java RMI. If a service depends on another service we need to add a proxy for this service in the userspace variation. Also a skeleton to the kernelspace has to be added as an endpoint for the proxy.

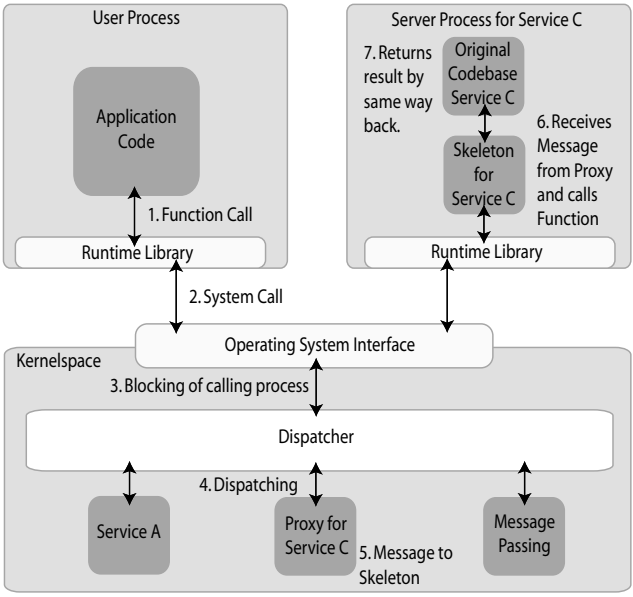

Figure 1. Migration of Service C from Kernelspace to Userspace

Using the regular scheduling mechanisms of a microkernel, the process containing the service has to wait until the scheduler activates it. This can lead to not predictable delay times which are depending on the used scheduling algorithm. In Real-Time Systems this delay needs to be predictable. To achieve this it is possible to use priority inheritance. The server process containig the services executing the system call, inherits the priority of the calling process. This leads to an immediate activation of the server process if no other process with higher priority is available. When the service is not needed a minimum priority can be assigned to the server process leading to an on demand activation of the service.

3.3 Context Switch and Virtual Memory access time

The microkernel architecture offers the possibility to separate the core of the OS (kernel) and services to provide a higher security by fault isolation. This separation comes at the cost of performance through complex context switches. Especially in embedded system the performance decreases rapidly when the context switch is not efficient and the applications make heavily use of system calls as the embedded systems are very restricted in ressources as computing power or memory. In our hybrid microkernel we want to minimize that tradeoff between security and performance. Therefore we use a similar approach as in

EMERALDS to realize an efficient context switch in a microkernel architecture and focus on the memory access time when using virtual memory in real-time systems.

EMERALDS enhances the context switch between a process and the kernel when a system call needs to be executed. Therefore the Kernel is mapped into the virtual address space of every process. That simplifies the context switch for a system call to the exchange of processor registers. Another advantage of that mechanism is that the kernel has direct access to the data of the calling process. This is very useful when a process wants to send data through a communication device. Nevertheless *EMERALDS* does not enhance the context switch between two processes. This leads to a big overhead in systems with multiple tasks where the time slices are pretty short. This is the reason why we also enhanced the context switch between processes.

To retain controll of the content of the TLB, we assume the use of a software managed TLB[2] for address translation inside the MMU[3]. When a context switch between two processes occurs the registers and the TLB needs to be filled with the entries of the next process. This can be very time consuming depending on the hardware architecture. In our case we tested our hybridkernel on a PowerPC 405 which has a software managed TLB with 64 entries. The time to restore the registers was determined to be about $1.5\mu s$ (see 3.4) and the time to write the 64 TLB entries was determined to be about $64\mu s$. This shows that the main overhead is caused by the writing of the TLB for the address space switch.

Another important issue in real-time systems is that every memory access could lead to a TLB Miss. TLB misses occur when there is no adress translation available in the TLB for the requested virtual adress. In that case the MMU has to replace an existing entry in the TLB with the needed entry from the virtual mapping of the process. In our case a TLB Miss causes a minimum overhead of about $1\mu s$ for writing the needed entry to the TLB. This overhead has to be considered in the WCET[4] and decreases the overall performance.

To guarantee an efficient context switch also between two process and a *O(1)* memory access time we make the restriction to the total number of TLB Entries needed by all processes (TLB_{system}) not to exceed the maximum number of TLB entries supported by the MMU (TLB_{MMU}). This guarantees that no TLB Entries need to be exchanged during a context switch and during runtime no TLB Miss can occur.

$$TLB_{system} < TLB_{MMU}$$

[2]Translation Look-Aside Buffer. TLBs are used to speed up the address translation process
[3]Memory Management Unit. The MMU translates virtual addresses into physical addresses using the TLB.
[4]Worst Case Execution Time. Needed for schedulability analysis in real-time systems

This also means that all processes of the complete system need to share the TLB. In systems with static page sizes this is pretty hard depending on the size of the pages (The smaller the harder). Some architectures offer the possibility to use memory segments with different sizes. The used PowerPC 405 MMU supports dynamic segments between 4K and 16MB. This allows to allocate a large block of contigous memory with only 1 TLB entry. To reduce the number of used TLB entries in the system the Intermediate-level Skip Multi-Size Paging algorithm [6] can be applied which is based on the well known buddy memory allocation algorithm [4].

3.4 Performance

As already stated the bottleneck of the microkernel architectures is the efficiency of the context switch which directly influences the performance of system calls. Therefore we determined the time for a NoOp system call by simply executing a large number of NoOp system calls from Userspace measuring the time until the last system call returns. The result of this is then divided by the number of systemcalls. As a result we get a mean execution time of $1.51\mu s$ for a NoOP system call on an Avnet Board with Virtex-II Pro (PowerPC405 with a clock of 200 MHz). A NoOp systemcall consists of two context switches. The mean execution time for a system call showed to be a good approximation of a single system call. We compared this mean execution time for a system call to the time of a simple NoOp function call within the userspace to show the overhead of the microkernel architecture compared to a library based OS. The needed time for a NoOp function call was $0.11\mu s$ beeing about 15 times faster than a system call. When converting this to cycles this is an overhead of about 280 cycles for a system call in comparison to a function call. If the service is located in the userspace two more context switches are necessary adding about another 280 cycles of overhead as we can assume that about 140 cycles are necessary for a single context switch.

Operation	Time	Clockcycles
NoOP Function Call (PowerPC405@200MHz)	$0.11\mu s$	22
NoOP System Call (PowerPC405@200MHz)	$1.51\mu s$	302

Table 1. Performance

Another thing to mention is the overhead through the configurability using SCL. We showed in section 3.1 that some code is inserted when configurable members are implemented. This code could lead to a small overhead every time the configurable member is accessed. In many cases this is not the case as the compilers today are pretty good in optimizing the code and removing code sections which are not necessary. Such an unnecessary code segment occur i.e.

in combination with the *hasMemberName()* method to check the availability of a configurable member. In the following example the presence of a scheduler is checked and the code would look like this:

```
if (hasScheduler()){
  getScheduler->schedule();
}
```

The compiler would remove the whole code if *hasScheduler()* returns false. Otherwise the resulting code would look like this

```
getScheduler->schedule();
```

4. CONCLUSION

We presented our concept of a Hybridkernel in which operating system components can be flexible configured in userspace or kernelspace. The advantage of this approach is that the same code base for the services can be used. Our well structured fine granular configurable operating system DREAMS allows an easy separation of system components with our configuration language SCL. This enables the application designer to choose the best tradeoff for his application between security and performance. Additionally we presented our fast concepts for high-performance context switches on the PowerPC405 architecture. Which enables the application designer to choice the more secure micro kernel approach for his applications.

The strength in security of the mikrokernel concept comes at the cost of performance. As we stated in this paper there is a limit in the capacity of the memory management units which lead to non deterministic memory access times if their capacity is exceeded. Our current approach will come to its limit if there is a lack of memory or the number of processes is growing. Therefore we will try to enhance the hardware support by using partially reconfigurable FPGAs. These FPGAs will then be used as an external MMU which will be highly adaptable to the needs of the system.

REFERENCES

[1] B. Behlendorf, S. Bradner, J. Hamerly, K. Mckusick, T. O'Reilly, T. Paquin, B. Perens, E. Raymond, R. Stallman, M. Tiemann, L. Torvalds, P. Vixie, L. Wall, and B. Young. *Open Sources: Voices from the Open Source Revolution*. O'Reilly, February 1999.

[2] C. Ditze. *Towards Operating System Synthesis*. Phd thesis, Department of Computer Science, Paderborn University, Paderborn, Germany, 1999.

[3] D. Hildebrand. A microkernel posix os for realtime embedded systems. In *Proceedings of the Embedded Computer Conference and Exposition 1993*, page 1601, Santa Clara, april 1993.

[4] D. E. Knuth. *The art of computer programming, volume 1 (3rd ed.): fundamental algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997.

[5] J. A. Stankovic and R. Rajkumar. Real-time operating systems. *Real-Time Syst.*, 28(2-3):237–253, 2004.

[6] S. Suzuki and K. G. Shin. On memory protection in real-time os for small embedded systems. In *RTCSA '97: Proceedings of the 4th International Workshop on Real-Time Computing Systems and Applications (RTCSA '97)*, page 51, Washington, DC, USA, 1997. IEEE Computer Society.

[7] A. S. Tanenbaum, J. N. Herder, and H. Bos. Can we make operating systems reliable and secure? *Computer*, 39(5):44–51, 2006.

[8] L. Torvalds. The linux edge. *Commun. ACM*, 42(4):38–39, 1999.

[9] A. Veitch and N. Hutchinson. Kea-a dynamically extensible and configurable operating system kernel. In *Configurable Distributed Systems, 1996. Proceedings., Third International Conference on*, pages 236–242, 6-8 May 1996.

[10] A. Veitch and N. Hutchinson. Dynamic service reconfiguration and migration in the kea kernel. In *Configurable Distributed Systems, 1998. Proceedings., Fourth International Conference on*, pages 156–163, 4-6 May 1998.

[11] K. Zuberi and K. Shin. Emeralds: a microkernel for embedded real-time systems. In *Real-Time Technology and Applications Symposium, 1996. Proceedings., 1996 IEEE*, pages 241–249, 10-12 June 1996.

[12] K. Zuberi and K. Shin. Emeralds: a small-memory real-time microkernel. *Software Engineering, IEEE Transactions on*, 27(10):909–928, Oct. 2001.