# Design and Implementation of the Lambda $\mu$-Kernel based Operating System for Embedded Systems

Kenji Hisazumi Teruaki Kitasuka
Tsuneo Nakanishi Akira Fukuda

Graduate School of Information Science and Electrical Engineering, Kyushu University
6-1 Kasugakouen Kasuga–city, Fukuoka 816-0922, Japan.
{nel, kitasuka, tun, fukuda}@f.csce.kyushu-u.ac.jp

## Abstract

*With large-scale embedded systems, improvement of development efficiency is one of the most important problems. In this paper, we design and implement an embedded operating system, called the Lambda operating system, which improves the maintainability and development efficiency of the operating system. The Lambda operating system employs micro-kernel architecture, which allows the operating system to be easily designed. In addition, we propose a method to improve operating system performance by reconstructing it in implementation. With the method, Lambda is implemented as monolithic kernel. The method allows the operating system to be quickly developed and gives high performance. This paper also shows that the method is useful through implementing a prototype of Lambda and its performance evaluation.*

## 1. Introduction

Since portable telephones and information electric home appliances come onto the market, embedded systems become more complex. Recent embedded systems have to provide functions such as multimedia data handling and network one. The development term of the large-scale embedded systems increases more than before. On the other hand, short-term development and time-to-market are required due to severe competition. Therefore, improvement of development efficiency is one of the most important issues.

Although maintainability and development efficiency of embedded operating systems are important issues, there are very few studies on them.

In this paper, we design and implement an embedded operating system, called the Lambda operating system, which improves the maintainability and development efficiency of the operating system.

Typically the maintainability, development efficiency and performance are exclusive. For example, an operating system which employs $\mu$-kernel structure is easy to develop. However performance of this one is bad. Most of embedded systems require high performance. On the other hand, an operating system which employs monolithic structure is faster than $\mu$-kernel structure. This structure needs careful developing to keep maintainability and development efficiency.

Therefore we propose transforming technique $\mu$-kernel structure to monolithic one to concomitants with high performance and maintainability.

## 2. Memory protections for embedded software

Most of embedded operating systems don't have any protection to keep the constraint of memory size, performance and so on. However memory protections become important since scale of software for embedded systems is becoming large. In this section, we consider the purpose of memory protections for embedded software.

There are some purposes of memory protections.

- For protection of text and data.
  To protect text and data from invalid memory access at runtime is necessary for the system to prevent into fatal.

- For security.
  We download software into embedded systems from the Internet. This software may destroy the embedded system software.

- For isolating untrusted software.
  When the developer buy an application program from other company (ex. WWW browser), this program may be unstable. This case is to protect other software from untrusted software.

- For debugging.
  Developing software may destroy other stable software area. If there is no memory protection, you cannot find this invalid memory access. If there is it, operating systems tell us it. To remove this protection after development is better.

The protection for debugging is especially interesting. Most of programs in embedded system are not changed after debug and we can trust them. In developing phase, we run programs under protection for debug. After development we remove protection of this part and get high performance. We consider of technique of removing protections next section.

## 3. Lambda Operating System

The Lambda operating system employs $\mu$-kernel architecture, which allows the operating system to be easily designed. Embedded systems have various hardwares and we must develop device drivers for them. This feature is very important for embedded systems. However, $\mu$-kernel architecture is slower and consumes more memory than monolithic architecture. Although these week points are improved by L4[5] roughly, but it is not enough. Cost and performance of embedded systems is more important than general–purpose systems. Therefore the Lambda operating system improves performance more, and cope with both performance and easy development by two features: the selectable memory protection and the automatic change from $\mu$-kernel structure to monolithic structure.

The first feature of the Lambda operating system is the selectable memory protection. The Lambda operating system provides a mechanism that allows processes of system servers and application programs to be implemented in either user process mode or kernel mode. In user mode, processes run with memory protection. In kernel mode, they run without it. At development phase, processes run in user process mode to debug the system. In this phase, we can use memory protection for effectively debugging. At final phase, memory protection can be removed. The system functions run in kernel mode. In this phase, system performance is higher than that in development phase.

Second feature of the Lambda operating system is the change from $\mu$-kernel structure to monolithic structure. A thread, which runs in kernel mode still needs inter–thread communications (ITC) to call other thread functions. Therefore system performance with this implementation is lower than that with monolithic structured implementation. The Lambda operating system alleviates the ITC overheads by automatically replacing the ITC mechanism to the function call mechanism. System performance with the function calls mechanism is higher than that with the ITC mechanism.

## 4. Changing $\mu$-kernel structure to monolithic structure

Now we consider a changing method from $\mu$-kernel structure to monolithic one. This method can be applied to only Remote Procedure Call (RPC). The Lambda operating system often uses RPC. RPC is generally used with small codes called stub. When a client calls a client stub, the client stub encodes arguments and sends a request to a server using RPC. A server stub receives the request, decodes arguments, calls a server function, and processes the request.

We implement a new stub generator called Lambda Interface Generator (LIG). It generates stubs and templates. If the server is in the same process of the client, the call from the client is performed by the function call. On the other hand, when the callee server is in a different process form the process of the client, its call is performed by using RPC rather than function calls (Figure 1). This method reduces communication overheads between a client and a server both of which are in the same process by replacing RPC to function calls.

## 5 Implementation and Evaluation

We implement a prototype of Lambda on a target machine of a Celeron 300A processor. We measure the execution time of some various models. Table 1 shows response times of basic system calls for reference. Figure 2 shows measurement models of same process RPC(a), cross process RPC(b), same process RPC with the changing method

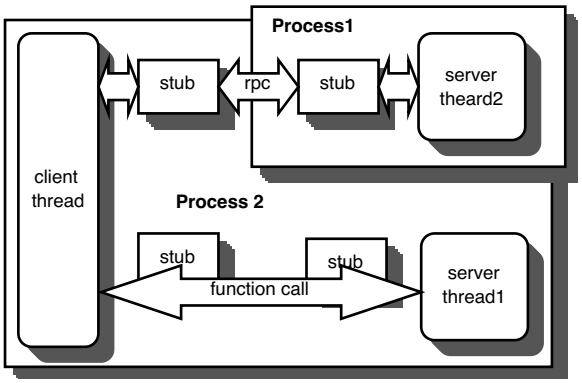


**Figure 1. Server stubs and client stubs**

**Table 1. Response time of basic system calls**

| system call | | cycles |
|---|---|---|
| thread_create | | 3630 |
| process_create | | 30000 |
| thread_sched | intra process | 411 |
| | inter process | 1986 |

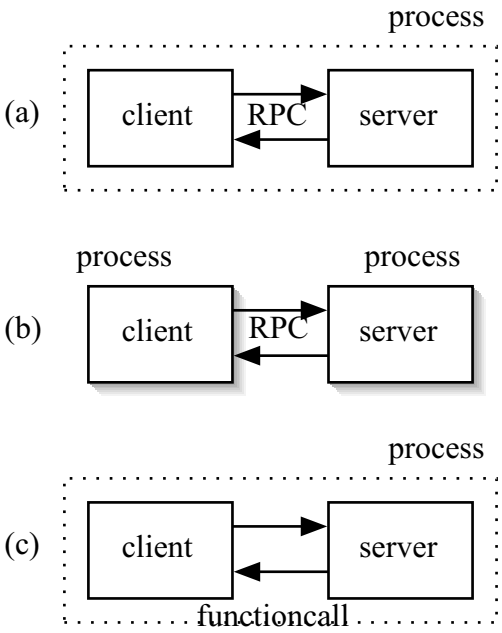


**Figure 2. The structure of measurement models**

of Lambda described before(called monolithic RPC) (c). Figure 3 shows the execution times of each model. The result of this measurement shows that the transformation of process structure improves RPC performance 20% and the changing method improves the system performance about 10 times.

## 6. Related works

LRPC[3] is a tequnique for reducing RPC overhead as same as our method. The LRPC reduces it dynamically. On the other hand, our method reduces it statically because of a tequnique for embedded systems. Our method allows to optimize globally with a compiler.

Component base operating systems, such as VEST[4] Knit[2] EPOS[1], improve descriptiveness with small object code and a good performance. However it is difficult to test a component. A component of the Lambda operating system can be run and tested with protections.

## 7. Conclusions

In this paper, we design and implement an embedded operating system, called the Lambda operating system. The

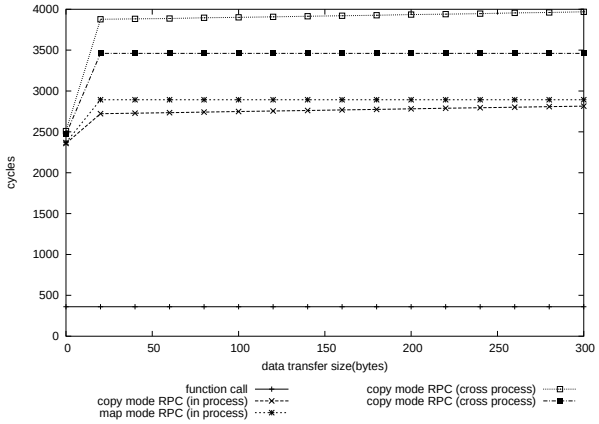


**Figure 3. The performance of cross process RPC, same process RPC and monolithic RPC**

Lambda operating system achieves both high maintainability and high development efficiency of the operating system. The Lambda operating system employs $\mu$–kernel architecture, which allows the operating system to be easily designed. In addition, we propose a method to improve operating system performance by reconstructing it in implementation. With the method, Lambda is implemented as monolithic kernel. The method allows the operating system to be quickly developed and gives high performance. This paper also shows the usefulness of our method through implementing a prototype of Lambda and its performance evaluation.

## Acknowledgments

This research is partly supported by Core Research for Evolutional Science and Technology (CREST) Program ”Advanced Media Technology for Everyday Living ” of Japan Science and Technology Corporation(JST).

## References

[1] A.A.Frohlich. Tailor-made operating systems for embedded parallel applications. In *Proc. of the 4th International Workshop on Embedded HPC Systems and Applications*, number 1586, pages 1361–1373, 1999.

[2] A.Reid, M.Flatt, L.Stoller, J.Lepreau, and E.E.Knit. Component composition for systems software. In *In proceedings of 4th Symposium on Operating Systems Design and Implementation*, pages 347–360. Usenix Association, 2000.

[3] B.Bershand, T.Anderson, E.Lazowska, and H.Levy. Lightweight remote procedure call. In *Proc of the 12th ACM symposium on Operating systems principles*, pages 102–103, 1989.

[4] J.A.Stankovic. Vest: A toolset for constucting and analyzing component based operating systems for embedded and real-time systems. Technical Report No. CS-2000-19, Dept. of Computer Science, Univ. of Virginia, 2000.

[5] J.Liedtke. Improving ipc by kernel design. In *Proc of 14th SOSP*, pages 203–205, 1993.