Resource Management for Embedded Systems

Roger Kreutz Immich, Diego Luis Kreutz and Antônio Augusto Fröhlich

Laboratory for Software and Hardware Integration

Federal University of Santa Catarina

PO Box 476 – 88049-900 – Florianópolis, SC, Brazil

{roger,kreutz,guto}@lisha.ufsc.br

Abstract

Classical strategies for resource management in operating systems are often complex and innapropriate for embedded systems. Implementations for these strategies may use either virtual function tables or long conditional structures to provide transparent access to different resources. This overhead is unacceptable for embedded systems. The EPOS operating system provides flexible and transparent access to resources for applications without incurring in unnecessary overhead. Metaprogrammed structures are used to predict, according to application usage and in compile time, whether a resource must use a polymorphic representation or may be accessed through direct calls. This way, virtual function tables are only used in the system when strictly necessary, and thus saving resources. In this article, we show that this strategy is a viable alternative for resource management in embedded systems.

Keywords: Resource Management, Static Metaprogramming, Operating Systems

1   Introduction

One of the main functions of an operating system is to manage hardware and software resources in a transparent and efficient way. General purpose systems often have to manage a great amount and variety of resources. Classical strategies of resource management in operating systems are thus often complex and dependent of application domain.

In order to provide application programmers with a reusable resource management application programming interface (API), for example, through a file system interface, general purpose systems often make use of conditional structures or virtual function tables in their implementation. The system doesn't know *a priori* what type of resource it must manage, and must provide access to all possible resources through a common interface. This causes the system to aggregate code blocks that may never be executed, but nonetheless will occupy system memory, and occurs in runtime overhead.

In embedded systems, applications typically use less resources than in general-purpose system. If, for example, an embedded application uses a single type of resource, an *application-tailored* operating system could provide management to that single resource, without incurring in runtime overhead or aggregating unnecessary code blocks. If an application, however, uses *n* resources, this operating system should also be able to provide a metamorphical, uniform interface for managing these *n* resources.

EPOS (Embedded Parallel Operating System) is an *application oriented* operating system that provides an adaptive, flexible and transparent interface for resource management[1]. Through the use of static metaprogramming techniques, and based on application analysis, it is possible to predict in compile time whether resources may be managed through a *direct call* or *polymorphic* interface. This way, only the absolutely necessary overhead is introduced into the system.

This paper elaborates on the resource management strategy in EPOS. Section 2 presents the EPOS metaprogrammed resource management framework. Section 3 evaluates the strategy, presenting a case study and evaluating overhead and performance of resource management in EPOS. Section 4 discusses related work. Section 5 discusses the results and finalizes.

2   Resource Management in EPOS

Resource management is a key point in operating system performance and usability. In the particular case of *application-tailored* operating systems [1], resource management is a specially interesting problem, as the application itself defines what resources must be managed in a given system instance, for a given execution environment. The application programmer must be provided with reusable interfaces, and with transparent component selection mechanisms.

One way to deliver transparent, adaptative resource management in an application oriented operating system is through the use of static metaprogramming. Metaprogrammed frameworks allow the system to select adequate components for resource management,
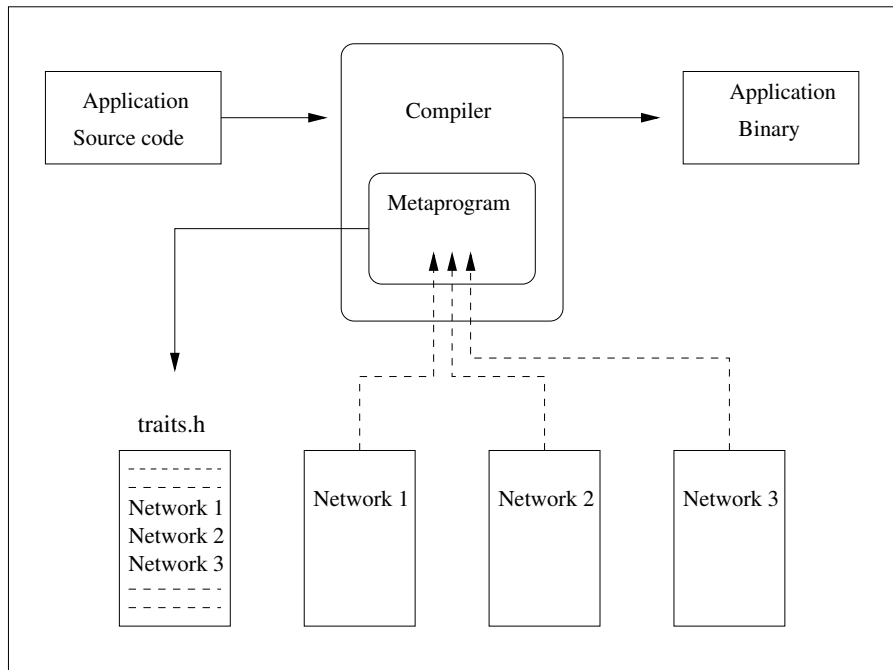
Figure 1. How the metaprogram works

generating flexible interfaces that may either generate *static* function calls or *virtual function tables*, according to application's needs and in compile-time.

EPOS relies on a specially developed metaprogrammed library to provide efficient and flexible resource management. Conditional structures (e.g. `IF-THEN-ELSE` and operators `EQUAL` are defined by this library, and implemented as described in [2]. These library functions are not restricted to resource management, and may be used elsewhere in the system.

Figure 1 illustrates the EPOS resource management resolution process which is applied in compile time. In the first step, macro components that satisfy user requirements are selected through application code analysis [3]. In the second step, specific, platform-dependent components are selected to become part of the system's final instance. In this phase, the metaprogrammed framework eliminates all virtual function call whenever possible, reducing final object code size.

As an example, if an application needs to use two network cards, the programmer simply declares two NIC objects. In compile-time, the application is analyzed according to the selected platform. The metaprogrammed framework identifies the types of network cards available in this instance of the system. If two network cards of different types are available, the resource management interface for these cards will present the same interface, and its implementation will be polymorphic. If two network cards of the same type are available, the resource management interface will still be uniform, but will provide direct access to the actual device, with no virtual function call overhead. This process requires no further

```
NIC nic0(0);
NIC nic1(1);

//thead 0
while(1){
    // ...
    nic0.send(BROADCAST, PROTOCOL, "A", 1);
}

//tread 1
while(1){
    // ...
    nic1.send(BROADCAST, PROTOCOL, "A", 1);
}
```

Figure 2. Sample Application

user interaction than to select and configure the target platform, and is transparent to the application.

Considering three hypothetical platforms "A", "B" and "C", and two network cards "X" and "Y". The programmer might use a similar code to the one presented in Figure 2 for different all three platforms. If that application were compiled for the hypothetical platform "A", with two network cards of type "X", the polymorphism of objects nic0 and nic1 will be replaced by direct calls to the actual instances of the network cards "X". The same process would be repeated for a supposed architecture "B", that has two network cards of type "Y". The great advantage is that the user application continues the same one, transparent and with a high degree of code reuse. On the other hand, the polymorphism could not be eliminated for platform "C", because, it makes use of one network card "X" and another network card "Y". In this

```
template <> struct Traits<PC_NIC>: public Traits<PC_Common>
{
    typedef LIST<PCNet32, PCNet32> NICS;

    static const unsigned int PCNET32_UNITS = NICS::Count<PCNet32>::Result;
    static const unsigned int PCNET32_SEND_BUFFERS = 8;
    static const unsigned int PCNET32_RECEIVE_BUFFERS = 8;

    static const unsigned int E100_UNITS = NICS::Count<E100>::Result;
    static const unsigned int E100_SEND_BUFFERS = 8;
    static const unsigned int E100_RECEIVE_BUFFERS = 8;

    static const unsigned int C905_UNITS = NICS::Count<C905>::Result;
    static const unsigned int C905_SEND_BUFFERS = 8;
    static const unsigned int C905_RECEIVE_BUFFERS = 8;
};
```

Figure 3. Description of the network cards in the traits file

in case, it is only possible to determine which calls goes to an specific network card in runtime.

2.1 Resource Management Metafunctions

A metaprogrammed `LIST` construct is used to generate a metalist in compile-time witch contains the available resources of a given type in the system. An configuration repository contains details regarding configuration and characteristics of all resources that may be used in a given platform. Figure 3 illustrates a sample traits configuration archive for the PC platform (Intel IA32 Architecture) in EPOS. In that example, configuration values for number of sending and receiving buffers are defined for three network devices: `PCNET32`, `E100` and `C905`. This information will be pertinent in compile time, during the definition of an instance of the respective component.

The `polymorphic` construct returns a boolean value obtained through the application analysis in compile-time using the previously defined metalist. When the system is configured with different devices of the same class (e.g. the PCNet32, E100 and C905 network cards defined in figure 3), this construct returns `TRUE` indicating that the use of the polymorphism will be necessary. When the metalist has only one element, or several elements with the same type, this function returns `FALSE`, indicating that the polymorphism can be eliminated and direct calls can be used to interact with the devices present in the list.

The `polymorphic` construct is used in a metaprogrammed conditional structure, that defines a `Base` variable (figure 4). The *Base* will be either a pointer for virtual methods (when polymorphic) or a pointer to an actual device (when not polymorphic), allowing direct calls to the device.

3 Evaluation

In order to test the efficiency of resource management in the EPOS, a simple application was implemented that sends the "A" character through the network interfaces. Two sample target configurations were use: one with

```
template<typename NICS>
class Meta_NIC {
  //...
  public:
  typedef typename IF<polymorphic,
         NIC_Base,
         typename NICS::template
                Get<0>::Result>::Result Base;
  // ...
};
```

Figure 4. Conditional Structures for Removing Polymorphism

a single network card, and another with two different types of network cards. This experiments demonstrates the percentage of resource management that is eliminated when virtual function calls are removed from the system.

We developed and compiled this application for the IA-32 architecture. Table 1 presents data and code memory sizes for test case A (concrete) and B (polymorphic). These values demonstrate that the resources management in the EPOS carried through in compile time, optimizes memory usage, allocating space only for the resources that really will be used in the application in question.

	Test Case A	Test Case B
.text	19308	19668
.data	88	88
.bss	432	432

Table 1. Size in bytes for the Test Case A (Single NIC) and B (Two Different NICs)

In a second experiment, we measured the access time to resources in the EPOS system. The measurements were taken by measuring the time immediately before calling

	Test Case A	Test Case B
Time	13.6	14.61

Table 2. Time in microseconds for access to a NIC in Test Case A and B

the `send` method, and when entering the actual `send` method in the NIC driver. We executed 100 iterations with 1000000 measurements each over a VMWare emulator in an Athlon64 3000 machine. Table 2 presents access time in microseconds for both test cases. These measurements demonstrate the efficiency of removing polymorphism whenever possible in a resource management strategy.

Through the use of the resource management strategies in EPOS it is possible to provide memory economy and to improve the access time to the resources. Memory economy is reached through the elimination of everything that is not be to the application execution, leaving the final code tailored to this application. The improvement in the access time to the peripherals is reached replacing, in compile time, virtual methods for direct calls.

4 Conclusion

The use of the static metapramming techniques to provide optimization in the resources management strategies for EPOS revealed a viable alternative for embedded systems. Despite the difficulties introduced by metaprogramming constructs, such as the increase of the complexity, difficulty of depuration and greater compile time, it was shown that that it is possible to use isolate its usage in the system only in the places where it becomes necessary, keeping the original structure in the others parts, continuing with the original flexibility and providing the necessary optimization. Current work focuses on further evaluating this techniques, and comparing it to other existing solutions.

References

[1] A. A. M. Froehlich, *Application-Oriented Operating Systems*, GMD - Forschungszentrum Informationstechnik, 1 edition, 2001.

[2] R. Robson, *Using the Stl - The C++ Standard Template Library*, Springer-Verlag, 2 edition, 1999.

[3] F. V. Polpeta and A. A. Fröhlich, "On the Automatic Generation of SoC-based Embedded Systems", in *In: Proceedings of the 10th IEEE International Conference on Emerging Technologies and Factory Automation*, 2005.