A Survey on Specific Domain Languages for Network Interface Cards

Apostolos Manolitzas

Abstract— **This article explains the main motivation of this project, presents the problem and discusses different approaches and solutions. Also it introduces a number of solutions that claim of solving the problems of writing device drivers.**

I. Introduction

FEELING the heat of their competitors, hardware companies create new products with a phrenetic pace. Those devices need the support of drivers which should be developed, debugged and tested in the most limited time in order to follow this pace. Although device drivers play the most critical part in the terms of performance, time pressure doesn't allow extensive testing. In combination with the assembly language, the low-level programming and the bits operations increase the error possibility and decrease the productivity. Not to mention the errant nature of that kind of programming.

We support that the most efficient solution to those problems would be the complete abstraction from the lower programming levels and the use of language, in which by providing the contents of the specification data sheet, a language generator should create the major part of the driver. This approach is not utopia, but requires intelligent, specialized compilers and rich libraries that would encapsulate more of the hardware knowledge. Some small steps have be made towards this approach and are presented in the article.

Furthermore, the different OS platforms provide another Sisyphean labor in writing device drivers, because the developer has to create multiple different instances of the same driver to support those differentiations. Attempts have been made to provide a unified interface that would encapsulate the different architecture of every device. From our point of view, we support that the solution will be given by using SDL(specific domain languages) with high level of abstractions, that could generate a specialized output adjusted even to the OS platform.

In this article we present analytically the main technological trends on writing high performance device drivers easily which are portable, easy to maintain and to reuse.

II. Related Work

Several approaches have tried to provide some solution to the problems that we mentioned before. Every approach has a different motivation and a particular range of applications. Some solutions was planed to provide an absolute solution to the complete set of device drivers and others are attempting to provide tools only for a small subset of drivers. Thus, there is no standard tool for writing device drivers, it means that the absolute solution haven't been found yet.

A.Manolitzas is a graduate student in the Department of Electrical Engineering in Columbia University of New York.

A. GAL

The GAL [3] (Graphics Adaptor Language) consists a novel attempt of using a specific domain language to support generation of device drivers by providing only device's description. In the description are specified the registers, the ports,the clocks and the most basic actions' sequence.

In their paper they describe in details the procedure followed and the issues that had to be dealed with so as to create a compiler generator specifically for device drivers. In general,they present the appropriate framework for the development of that type of software.

In order to support their idea, they implement a driver for graphics devices of the S3 chips series using their language. The generated driver is compared with the current hand-crafted implementation of the S3 drivers. In the results of the comparison they point out that the performance of their driver surpasses in performance the original, implemented in C, driver in some cases. In addition they claim that their code is very easy to write, to maintain and to reuse it. Justifying the essentiality for code reuse, they present the argument that code development is mostly redevelopment and the new code is a modification on the existing one adjusted to new requirements. With their language this can be done easily.

Another interesting idea, that worths exploring, is compilation in several independent stages. On every stage, additional information can be added to finalize their output. But the magic of the conception involves the differentiation of information that can be added to produce an output dedicated to a specific problem. This approach is called partial evaluation and it is a method which evaluates parts of a program in advance so as the code generated will be more efficient and dedicated to a specific application.

We support that approach and our research is based on the procedure described in their paper. We believe that writing device drivers should be more efficient, with less pain and the code should be easy to reuse so as to increase productivity. Such an approach is showing the path to solution. The carriage to that solution is the specific domain languages.

But GAL is not panacea. It is too specialized and requires much effort to support that specialization. The authors presented only the creation of device drivers for a small group selected chips of the same series. That visual-

ize the difficulty of the approach.

B. Devil

The same research team, identified the problems of the previous solution and they decided to solve them with a more generic language that would apply to the entire range of drivers. So the proposed Devil [1], is a language or better a supplement tool for the developer that would help him avoid common mistakes and increase his productivity.

They claim that they created a compiler that checks safety critical properties and generates robust hardware operating code. They used Interface Definition Languages (IDL) to describe the hardware and its functionality. It provides the programmer with abstractions and syntactic constructs that are specific for describing devices. Particularly, Devil is a compiler that generates automatically stubs which provide an interface to the device. The interface is mostly consisted of macros and nicely defined registers. Such an approach allows logical error checking during the compilation in order to avoid the tedious real-time debugging.

One of their main consideration was the property verification. They support the principle that a large portion of errors are caused be mistyping, wrong type matching and type overlapping. Their argument is completely justifiable, because most drivers actions are interaction with the hardware which involves reading and writing to device registers. Those actions are pure bit operations which are high errant actions.

The paper insists on the necessity of fast error-detection and compiling checking. To support the validity of their language,they rewrote the device drivers for an IDE device, for a mouse and for a NE2000 Ethernet. They presented some comparisons based on error evaluation between the rewritten code with the use of Devil and the hand crafted-code. The results show that the possibility of error is reduced with the use of Devil. Specifically, the tests are based on the mutation analysis technique, which main objective is to expose errors that occur by inserting, replacing or removing a character from a token. Another remark is for the performance of the driver that remains as high as the hand-crafted drivers.

In conclusion we can say that this language provides a formal way to define safely macros and registers. Of course the attempt can be characterized more the positive in terms of properties checking, but defining macros and registers is a technique that a decent, self-respect developer does. Also they didn't deal at all with the problem of multiple operating system portability. We believe that this approach is helpful but doesn't provide a clear total solution.

C. UDI

Another critical issue that device writer faces is the portability of the driver to the different OS. There are many differences among current operating systems that influence the environment for device drivers and other kernel modules. Some support kernel threads; others do not. Some support preemption; others do not. Some support dynamically loadable kernel modules; other do not. Variations in memory management and synchronization models also impinge upon the device driver environment.

In order to support that difficulty, UDI [2] (Uniform Driver Interface) provides a common framework for drivers of many different types. This makes it easy to write new types of drivers, since all drivers share a common look and feel. It also allows flexibility in hardware and software configurations, since any driver can potentially communicate with any other driver.

Certainly, this approach is a solution to the portability problem but requires programmers cooperation. The main disadvantage is that every developer should be responsible for following that environment. Also that environment doesn't provide him type checking and type verification as the other attempts did. UDI focuses mainly, only on the high level part of the drivers and their interaction with the operating system.

D. Flick

In this paper, they suggest a compiler, named Flick [4] that is used for compiling IDLs. Although,it has nothing to do with device drivers, they analyze a very interesting technique that should be used in our project.

IDL compilers, generally, generate stubs and skeletons, which are by their nature header files, with limited functionality. The stubs provide an interface for method invocation. Usually, stubs contain the mapping of the IDL to the desired language as C,C++, JAVA,etc. Their mapping is based on simple rules with no intelligence, optimization or efficiency.

In their proposal, they suggest the use of common compilation techniques to optimize the generated code. That type of compilation is consisted by three phases. In each phase they create a representation of the interfaced defined by the IDL input. This representation contains a different level of specialization and it's completely independent from the previous stages . So for example, the first level is just an abstract representation and the last the code specialized for CORBA requests with TCP/IP.

Following this idea, the device driver compiler should have the qualifications of creating an initial abstract representation of the driver and in every step to add special components, eventually from a large library of code, that would lead to the generation of code specialized for a certain chip and OS.

E. Tools

Several others attempts have been made to make easier the life of the device driver programmer. Libraries [6] have been built and Toolkits like WinDK [5] have been created, but they don't deal with the whole range of the problem. A very prominence attempt is a commercial product called jungo [7]. By plugging the device hardware in a computer slot, their diagnostics scan it and they create a framework based on the characteristic read. Following,the developer provides the specification of the device through an interactive dialog and finally the program generates the driver

for the device. Of course this is a very limited environment without giving any freedom to the developer.

III. SDL for network cards

Through the survey we found some exciting ideas, but our belief is that only by defining an SDL language for every device family, we will manage to solve the largest portion of the problem. Following this essential idea, we are going to propose a specific domain language for network interface cards.

The methodology of our research follows the commonality analysis. Based on that method we are going to define network cards families and network cards operations. In practice, commonality analysis is done by experts and experienced developers, but within the borders of this attempt we will go through the analysis for the drivers of one operating system. Our study will focus on ISA and PCI network cards, and their corresponding Linux 2.4.10 drivers.

Briefly we can identify 3 patterns which appear in the existing drivers that could be used for our guideline: The operation pattern which is code fragments that are repeated in the driver and differs only by data, the combination of operation patterns which are combination as operations and the last is the control pattern in which a decision is made concerning which operation pattern to be used.

Hopefully, based on those patterns, we expect to define the specific domain language.

References

[1] Fabrice Merillon,Laurent Reveillere,Charles Consel,Renaud Marlet, Gilles Muller. Devil: An IDL for Hardware Programming. *OSDI 2000*, pages 17-30, San Diego, October 2000.

[2] Project UDI. *UDI Specifications, Version 1.0*, September, 1999. URL: www.project-udi.org.

[3] S. Thibault, R. Marlet, and C. Consel. Domain-specific languages: from design to implementation - application to video device drivers generation. *IEEE Transactions on Software Engineering*, 25(3):363-377,May-June 1999.

[4] E. Eide,K. Frei,B. Ford,J. Lepreau,G. Lindstrom. Flick: A Flexible, Optimizing IDL Compiler. *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation* (PLDI), Las Vegas,NV, June 15-18,1997.

[5] BlueWater Systems, Inc. *WinDK Users Manual.* URL: www.bsquare.com

[6] Compuware NuMega. *DriverWorks User's Guide.* URL: http://www.compuware.com/products/numega/drivercentral/

[7] Jungo Ltd. *WinDriver V5 User s Guide.* URL: www.jungo.com