Language-based Approach for Software Specialization

Tegawendé F. Bissyandé

Language-based approach for Software Specialization

Tegawendé F. Bissyandé
LaBRI, University of Bordeaux
France
Email: {bissyand}@labri.fr

***Research proposal*—Research advances in electronics have lately enabled the deployment of devices with various capabilities. These equipments are more and more used as basic entities integrated in larger systems where their functional autonomy is highly appreciated. Such systems, present in most infrastructures of our daily environment, are widely used in various fields such as mobile telephony, automobile industry, aeronautics, domotics and recently medicine.**

Applications running in these systems must interact with their environment so as to offer services that are better suited to the evolving context. To do so, they benefit from the capacities of middleware which, thanks to the high level services offered, ease the application development by hiding hardware, protocol and system heterogeneity.

Besides, embedded systems are much more constrained in terms of memory, energetic and computing resources. Furthermore, applications in embedded systems must often fulfil specific requirements for soundness and real-time capabilities. Existing commercial off-the-shelf middleware are therefore not suited as is to all the needs stated above. Yet designers of software for embedded systems increasingly use general-purpose off-the-shelf middleware/libraries to provide sophisticated functionalities while meeting time-to-market and reliability requirements.

Our project aims at planning out a novel approach to automate the specialization of software according to the needs of the applications that will make use of them. Practically, we suggest a tight coupling between applications and middleware for a deployment time specialization that takes into account the application needs and the execution context.

We are carrying this study at the LaBRI laboratory in Bordeaux, France under the supervision of Laurent Réveillère. To give a better overview of the kind of issues we plan to address, we summarize in the following preliminary results[1] obtained during our master thesis internship. A short paper on this work is under submission for publication in IEEE Embedded Systems Letters.

Summary of Current work

In recent years, software developers have been producing more and more sophisticated software to be embedded in every device incorporating a processor. These developers must meet a number of constraints, including robustness and the need to reduce cost and time-to-market. One solution to meet these constraints is to use off-the-shelf libraries. Nevertheless, such libraries are often general purpose, and contain many features that are not needed by a specific application. Their use in the final deployed software thus incurs significant - and unnecessary - overhead in the memory footprint of the software. While ROM sizes are frequently higher than RAM sizes, considerations such as space, weight, power consumption and price imply that limiting both the code and data memory usage of an embedded system is critical.

In the world of desktop computing, the mix of applications varies frequently, and these applications vary significantly in their functionalities. Thus, in this setting, it is useful for libraries to provide many features, that can be shared among different applications in various permutations. On the other hand, in embedded systems, such as routers, washing machines and coffee makers, the set of applications is typically fixed, or varies extremely rarely. This makes the generality of libraries cumbersome in the final software although this property can be required in the design phase to promote code reusability (see the design of PURE [7]). A solution to reduce the memory footprint in the embedded system setting is then to specialize each *shared library* used in an embedded system with respect to the requirements of the set of applications that make use of its functionalities.

We propose an approach to automatically specialize libraries at the source code level according to the needs of a set of applications. Our approach works in two steps: first it identifies the functions of a library that can be called directly or indirectly from a given set of applications, and then it removes the implementations of all other functions from the library. A similar strategy is applied to data structure fields, to further reduce the memory usage. This approach is complementary to most previous code compaction techniques [2], [4], [5], which are designed to be applied to compiled code. In particular, we find that we obtain a better rate of compaction by combining our approach with the compaction provided by gcc than what is achieved by either technique alone. Our approach can also be used for code understanding and easier debugging, as the application developer is no longer faced with library code that is not relevant to the considered applications.

Our approach: We propose a specialization process that is carried out without any intervention from the application programmer in the form of annotations or similar techniques. Our compaction tool `SpecTool` removes from the library's code all the functions and data (statements and fields of data structures) that will not be needed by any of a given set of applications.

`SpecTool` performs the specialization in three steps. In the first step, an analyser collects for each application information about its use of the functionalities provided by the library. In the second step, the collected information is merged to define

[1]This is a joint work with Laurent Réveillère, Julia L. Lawall and Gilles Muller

library file	*original runtime system*			*customized runtime system*			*reduction rate*		
	LOC	`gcc`	`gcc + strip`	LOC	`gcc`	`gcc + strip`	LOC	`gcc`	`gcc + strip`
libz2zrt.so.0.0.0	3,751	172 Kb	48 Kb	2,611	128 Kb	36 Kb	**30.4 %**	**25.6 %**	**25.0 %**
libz2zrt.a		236 Kb	68 Kb		184 Kb	48 Kb		**22.0 %**	**29.4 %**

TABLE I: Specialization of z2z runtime for the tunnel of SMTP over HTTP (`LOC := Lines of Code`)

a unified usage signature of the library. This global signature is then used to generate a set of transformation rules [8] matching code fragments that can be safely removed from the library. Finally, in the third step, the Coccinelle[2] source code transformation engine applies the rules to automatically generate a specialized version of the library.

Case study: Our work was motivated by the issues of memory footprint encountered in the z2z project [3]. Z2z provides a generative approach to network protocol gateway construction to address the problem of protocol incompatibility in a domotics environment. Z2z gateways rely on two kinds of libraries: the z2z runtime system, which is shared by all z2z gateways, and external libraries, which provide functionalities such as parsing that are specific to a given protocol.

The z2z runtime system provides a number of low-level network-related functionalities that are common to a range of protocols. These include support for a wide variety of network transport protocols (e.g, UDP or TCP), communication modes (e.g., synchronous or asynchronous), and protocol types (e.g., binary or text-based). Nevertheless, any given gateway implementation is not likely to use all of these functionalities. Deploying the complete runtime system with one or even several gateways on an embedded system will therefore require more memory than necessary. For instance, the z2z implementation of the SMTP/HTTP tunnel is composed of two distinct gateways (SMTP to HTTP and HTTP to SMTP) that use the z2z runtime system differently. Table I shows that we achieve an average reduction rate of 25% for the static and dynamic libraries in this setting.

There are also opportunities for footprint reduction in the case of external libraries used by a z2z gateway. Network message parsing is not provided by z2z, and thus it is necessary to use an external parser. Libraries including parsers already exist for many network protocols. For example, the camera gateway developed with z2z to enable a SIP [9] based telephony client to receive images from an IP-camera accepting only RTSP [10] for negotiating the parameters of the video session, uses an external library, *oSIP*,[3] to parse SIP messages. However *oSIP* includes not only a parser, but also all the functionalities that an arbitrary SIP application may need. Therefore, including the entire library in the runtime program significantly increases the gateway memory footprint. Because of the internal dependencies of the library, it is difficult to safely extract by hand the minimal fraction of code that is required.

Related Work: There has been much attention paid to code-size reduction, especially in the context of embedded systems. The techniques available in the literature include two main families extensively reviewed in [2]:

- Lossless *compression* techniques [6], which are applied to program code in order to produce an equivalent but smaller representation.
- Compaction techniques, that were described by Bell *et al.* [1] as *irreversible compression* techniques. Most compaction techniques are directly applied to native code or produce executables from source code.

The main drawbacks of compression techniques lie in the fact that they can introduce processing delays since the compressed code must be decompressed before execution.

Native code compaction techniques are usually based on classical compiler optimizations, such as elimination of unreachable code, dead code and redundant code inside a program [4], [5]. Our tool instead performs a specialization according to the needs of a collection of unrelated programs. It is noteworthy that our source-code compaction technique is not incompatible with the other techniques. Indeed, since our work is applied to the source code, the compiled libraries can always be reduced again and/or optimized with efficient compiler infrastructures such as the *Low Level Virtual Machine* (LLVM).[4]

[2] http://coccinelle.lip6.fr/

[3] http://www.gnu.org/software/osip/

[4] http://llvm.org

REFERENCES

[1] T. C. Bell, J. G. Cleary, and I. H. Witten. *Text compression*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[2] A. Beszédes, R. Ferenc, T. Gyimóthy, A. Dolenc, and K. Karsisto. Survey of code-size reduction methods. *ACM Comput. Surv.*, 35(3):223–267, 2003.

[3] Y.-D. Bromberg, L. Réveillère, J. L. Lawall, and G. Muller. Automatic generation of network protocol gateways. In *Middleware '09: Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, pages 21–41, Urbana Champaign, IL, USA, 2009. Springer-Verlag New York, Inc.

[4] K. D. Cooper and N. McIntosh. Enhanced code compression for embedded risc processors. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 139–149, Atlanta, GA, USA, 1999.

[5] H. He, J. Trimble, S. Perianayagam, S. Debray, and G. Andrews. Code compaction of an operating system kernel. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, pages 283–298, San Jose, CA, USA, Mar. 2007. IEEE Computer Society.

[6] C. Lefurgy, E. Piccininni, and T. Mudge. Reducing code size with run-time decompression. In *Sixth International Symposium on High-Performance Computer Architecture (HPCA-6)*, pages 218–228, Toulouse, France, Jan. 2000.

[7] D. Lohmann, W. Schrder-Preikschat, and O. Spinczyk. On the design and development of a customizable embedded operating system. In *Proceedings of the International Workshop on Dependable Embedded Systems*, pages 1–6, Florianopolis, Brazil, October 2004.

[8] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller. Documenting and automating collateral evolutions in Linux device drivers. In *EuroSys 2008*, pages 247–260, Glasgow, Scotland, Mar. 2008.

[9] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session Initiation Protocol. RFC 3261, Internet Engineering Task Force, June 2002.

[10] H. Schulzrinne, A. Rao, and R. Lanphier. Real Time Streaming Protocol (RTSP). RFC 2326, Internet Engineering Task Force, Apr. 1998.