International Journal on Advanced Science Engineering Information Technology

Vol.8 (2018) No. 4-2
ISSN: 2088-5334

A Program Optimization Method for Embedded Software Developed Using Open Sources

Sang-Young Cho[#]

[#] *Division of Computer & Electronic Systems Engineering, Hankuk University of Foreign Studies, Yongin, Gyeonggi, 17035, Korea*
E-mail: sycho@hufs.ac.kr

***Abstract*— Program optimization or software optimization is the process of modifying a software system to make some aspect of it work more efficiently or use fewer resources. When developing software for embedded systems, open source libraries are usually used. An Embedded software built using a variety of open software and libraries is apt to have too many unused codes because open source libraries contain many functions and features to be used in various applications. In this paper, we describe the process of a library optimization method to reduce memory consumption for the user interface software of a set-top box product. The overall optimization process uses freeware tools developed for the Linux operating system. We devised an optimization technique to compensate for operation imperfections in the target system. The original Qt library of 19.57 MB was optimized to be 7.26 MB in program image size. In the case of the DirectFB library, 3.2 MB was reduced to 2.4 MB. The optimization process can be applied to any embedded software that are developed with open source libraries or sources.**

***Keywords*— software optimization; open source library; static and dynamic analysis; freeware tool.**

I. Introduction

Program optimization or software optimization is the process of modifying a software system to make some aspect of it work more efficiently or use fewer resources. In general, software engineers have been trying to optimize computer programs so that a program executes more rapidly, is capable of operating with less memory storage or other resources, or consumes less power. Usually, the performance optimization has been the most focused target for program optimization and there are many researches in various optimization layers: design, algorithm and data structures, source code, build, compile, assembly, and run-time [1].

Due to the proliferation of personal mobile devices and sensor-based embedded systems [2], minimizing power consumption is one of the primary challenges that today's developers face. Some research investigated the methods to reduce power consumption using software optimization because software plays an important role in device energy efficiency [3].

In the perspective of memory usage, dead code elimination has attracted a great attention in compiler theory for removing codes that do not affect the program results. Removing such code has several benefits: it shrinks program size and it allows the running program to avoid executing irrelevant operations, which reduces its running time. It can also enable further optimizations by simplifying program structure [4].

When developing software for embedded systems, open source project libraries are usually used. For example, Qt is a cross-platform development framework consisting of a tremendous size of libraries [5]. ffmpeg provides many codecs and application tools for multimedia processing and is used in various multimedia applications [6]. These well-known open source libraries are well-modularized and have a lot of functions. However, the associations between modules and the usage scopes of modules for an actual application software development are not precisely defined. This incurs that embedded software developers use the whole set of libraries for developing application softwares.

Embedded software built using a variety of open software and libraries may have many unused code since open source libraries are aiming to be used in various applications. They usually contains too many unused functions for a specific application. This leads to many code units (e.g. packages, classes, methods, functions) that the running application never uses when the embedded software contains the whole library. Therefore, memories for storing and executing the program are wasted due to the unnecessary codes.

After developing a prototype of an embedded system product, the prototype needs software optimization even thogth the desired functions of the prototype operate well because memory cost is very sensitive to the unit price of a commercial product. Also, unused codes deployed in a product units have an undesired impact on software operations when targeting a constrained infrastructure. Some devices may constrain applications due to restrictive hardware resources such as small primary or secondary memories [7]. To minimize memory usage, we should remove unused modules, files, functions, and variables. However, this is not a simple task for large libraries.

A common method for library optimization of a prototype software is to identify and remove dead codes through static and dynamic analysis [8]. Static analysis approaches for dead code elimination make use of the static information of a program to select the minimal subset of used program elements. Static analysis obtains information about the class hierarchy or function call graph at the source level by analyzing the library structure and the usage of the library in the application. Static analysis cannot obtain the call graph information of dynamically linked functions [9-10]. Static analysis is not applicable efficiently on dynamic languages with no static type declarations i.e., some code units whose usage cannot be anticipated by these techniques will stay even if they are really not used during the application execution. Dynamic analysis gathers program execution information while the program is executing on a target system. It provides information (i.e., execution flow, alive objects, execution statistics) about the program codes actually used. By using code analysis information, it is possible to remove dead codes of unused modules, files, functions, and sentences [7]. Curretnly, there is no integrated tool to optimize libraries in an overall view of both static and dynamic analysis information. There is a tool to facilitate algorithm development optimization in a specific area [11]. Recently, [12] proposed a practical library optimization method for open source libraries using freeware tools and this paper is an extension of the work of [12].

In this paper, we describe a software optimization strategy for resource-constrained embedded systems built using open source libraries. Especially we focus on minimization of program footprints of graphic user interface (GUI) software of Set-top Box (STB) products on primary and secondary memories. In our method, the overall optimization uses freeware tools running on Linux and includes optimization techniques to compensate for imperfections of the target system and instabilities of freeware tools.

The rest of this paper is organized as follows: Section II describes the target system for optimization and explains the freeware tools used during optimization process. Section III describes the opmization objectives and the details of the optimization process. Finally, we conclude in Section IV.

II. Material and Method

In this paper, we propose a method to minimize the primary and secondary memory usages by optimizing open source libraries used in the GUI part of STB that is limited in memory resources.

A. *Software Optimization Environment*

Fig. 1 shows the target STB hardware board. The motherboard is equipped with an application processor of MIPS core and has 256 MB of DRAM and 128 MB of NAND Flash. In addition to the motherboard, a TV receiving module, a conditional access system (CAS) module, and a serial conversion module are provided. The target board is connected with a development computer with the serial module. The TV receiving and CAS modules are not used during software optimization.

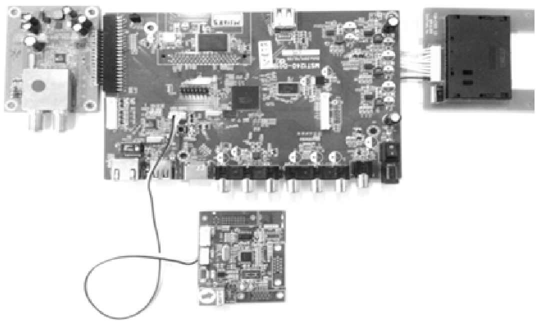

Fig. 1 The target hardware board of Set-Top Box

Fig. 2 shows the software layer structure of the target STB system. There is a STB platform consisting of the Qt library and a proprietary SDK on the Chipset platform provided by the vender that provides the STB reference board, and based on this, there is the STB GUI program at the application level including the MythTV library. The Qt and DirectFB [13] libraries are targets to be optimized.

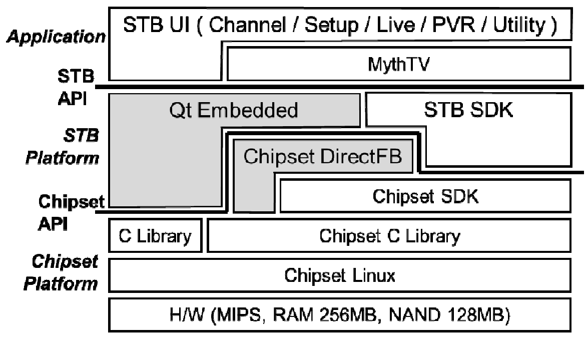

Fig. 2 The software layer structure of the target STB system

Software or library optimization strategy and steps depend on whether the structure and internal details of the software system is informed in advance or not. If a software developer of the prototype product is optimizing by himself/herself, since he/she has knowledge about the development environment, the application program, and the libraries used, the developer can go straight ahead after learning how to use optimization tools. Otherwise, a development and optimization environment should be established and the target application program and libraries should be investigated in detail. In this paper, we describe the strategy of software optimization by assuming we have no prior knowledge of the prototype system and software.

In order to optimize the libraries used, it is necessary to understand the structure of the library and to analyze the used and unused parts of the libraries when the application is executed on the system. Also, the optimization direction should be set by confirming the constrains of resources. Based on this, the redundant source should be removed and simplified. In addition, the correct operation of the modified program should be verified and confirmed.

The optimization steps are divided by three steps. The whole structure of target software and related libraries are studied at the first step. In this step, we use several static and dynamic analysis tools and become familiar to the optimization environment. Based on the static and dynamic analysis information of the target software, we perform optimization at the second step. The optimization starts from library modules and go into detailed function levels. Finally, we refine the optimization with the status analysis tools. During optimization process, some techniques are devised to overcome obstacles that occur without a priori knowledge.

B. *Related Optimization Tools*

The target system was built on the Linux operating system running on a MIPS core. The GUI software of target system has been developed using open libraries that companies can freely use. To optimize the software, we should select appropriate tools among the freeware tools available in the Linux operating system. The tools used for optimization are largely divided into static analysis tools, dynamic analysis tools, and status inspection tools. Table 1 summarizes the open source tools and commands used for library optimization.

TABLE I
TOOLS USED FOR SOFTWARE ANALYSIS AND OPTIMIZATION

Tool type	**Name**	**usage**
Static	Source Navigator	source structure
	Doxygen	class diagram, call graph
Dynamic	Gprof	function usage, call graph
	gcov, lcov	line, branch, function coverage
Status	pmap	memory usage
Command	ldd	dependency of shared library
	nm	Symbols of object files
	readelf	ELF file information

Analyzing the source itself without running the program is called static analysis. Static analysis tools allow you to understand the structure of the program and provide unused code information through dependency analysis. Doxygen is the de facto standard tool for generating documentation from annotated C++ sources [14]. In particular, Doxygen automatically generates class dependency graphs, inheritance diagrams, and inclusion relationship graphs. In this paper, we use Doxygen and Source Navigator to understand the class hierarchy of application programs and libraries.

Dynamic analysis tools extract run-time information such as the execution times or execution numbers of functions, lines, branches, and function call graph. The tools usually gather run-time information by inserting analysis codes into the application program or by extracting system data maintained by operating system [15]. gprof collects information such as execution time, time occupancy, and calling relation of functions executed in the program. This tool provides profiling information just about functions and we cannot use the tool to get run-time information of line or branch of program code. This tool can be effectively used to optimize the program because it can obtain the information of the functions that need to be optimized for improving the performance of the whole program. gcov and lcov are tools that can measure the coverage of a program, and can measure the coverage of lines, branches, and functions. That is, one can get information about which lines in the program are being executed, which functions are called, and where to branch from statements such as if-else or switch-case. Unlike gprof, we can get line and branch execution information and, by adding option, coverage information of dynamic library. However, we cannot get information such as function call relation and execution times of functions. gcov and lcov are useful tools for removing unused code.

The Linux kernel continuously updates and stores various status information for each running process under the /proc/${pid}/ directory. From these information stored in the directory, the Linux pmap command properly outputs information about the memory map in a human-readable form [16]. With pmap, we can figure out what dynamic libraries are mapped to the primary memory that is used by a particular process and the size of each mapped library. We can also observe changes in the program's behavior by observing the changing memory usage while the target software is running.

In case of binary files, there is no program source, so we cannot use static tools to investigate program dependencies. However, because the binary file contains information about the program or library involved, we can use the Linux system commands to obtain program dependency information. ldd shows information about all shared libraries that depend on the binary file [17]. nm is used to examine binary files (including libraries, compiled object modules, shared-object files, and standalone executables) and to display the contents of those files or meta information stored in them, specifically the symbol table [17]. The readelf command allows us to view information stored in the ELF header of an ELF file [18]. This information is usually a list of classes or functions that an ELF binary file has or is using. By analyzing the function dependency of a program by using the static tools and the Linux system commands, we can have information about unnecessary modules, classes, and functions in the upper level.

III. RESULTS AND DISCUSSION

In this section, we will describe the details of the optimization progress and discuss the results at each optimization step.

A. *Initial Software Status*

Table 2 shows the initial software size of STB application. The program size of each module is measured as static, i.e., NAND flash storage consumption.

TABLE II
SOFTWARE SIZE OF EACH MODULE OF TARGET SYSTEM

Storage	**Module**	**Size**	**Explanation**
NAND Flash (128MB)	Kernel	2.5 MB	Linux
	Rootfs, Lib	15 MB	Busybox and library
	DirectFB	3.2 MB	DirectFB, Fusion, Font
	QT	19.6 MB	QT library
	APP	23 MB	Application, library Font, Images, XML

We are targeting the QT and DirectFB libraries for optimization because Kernel and Rootfs are given as binary files by a vendor and the GUI application may be developed model by model. Among the two optimization target libraries, Qt is frequently used in the GUI application and its size is dominant among the used libraries. We need to focus on Qt during optimization process. Table 3 shows the initial size of each important class category of Qt library of the version 4.8.2. We can see the GUI component classes and Core classes are dominant in size and they are the main targets for optimization.

TABLE III
THE SIZE OF EACH CATEGORY OF THE QT LIBRARY

Module	Size	Explanation
libQtCore.so	3.6 MB	Core non-graphical classes
libQtGui.so	12 MB	GUI components
libQtNetwork.so	1.2 MB	Network programming
libQtSql.so	254 KB	DB integration using SQL
libQtTest.so	151KB	Unit testing tool
libQtXml.so	288 KB	XML handling class

B. Optimization Process

For the first step of the optimization process, we surveyed the Qt and DirectFB libraries and explored the optimization tools to be used. We also constructed the development environment. We referred to the homepage of each library and the documents on the Internet to understand the overall structures of the libraries. We searched and studied the tools to be used for optimization. Fig. 3 shows the detailed optimization steps that we performed.

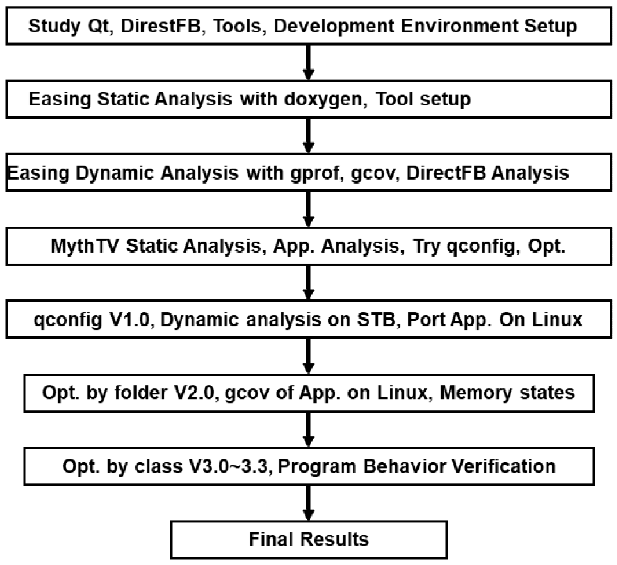
Study Qt, DirestFB, Tools, Development Environment Setup
Easing Static Analysis with doxygen, Tool setup
Easing Dynamic Analysis with gprof, gcov, DirectFB Analysis
MythTV Static Analysis, App. Analysis, Try qconfig, Opt.
qconfig V1.0, Dynamic analysis on STB, Port App. On Linux
Opt. by folder V2.0, gcov of App. on Linux, Memory states
Opt. by class V3.0~3.3, Program Behavior Verification
Final Results

Fig. 3 The optimization steps for STB UI libraries

We analyzed the source codes of the libraries using Source Navigator to understand the internal flows of important operations. We also obtained the static analysis data of the Qt and DirectFB libraries using Doxygen We used the Easing sample application of Qt library as an initial study target for static and dynamic analyses. These steps clarified the internal structures and the class usages of the libraries.

We found that the dynamic analysis tools, gprof and gcov, did not work on the target system because the development environment of the system had some bugs. To get the dynamic analysis information of the Qt library, we have also run the Qt example program, Easing, on a normal Linux machine with the gprof and gcov tools to obtain indirect dynamic analysis data of the library. Through this process, we became familiar with the development environment and tools, understood the structure of the Qt and DirectFB libraries, and prepared for actual code optimization.

In order to optimize the Qt library used in a real STB application, it is necessary to analyze the GUI program at the application level. Through a static analysis of the GUI program, the program is implemented based on a class called MythScreenType, and this class has been found to inherit QObject. Qt's utility classes are used directly at the application level. Based on this result, we need to analyze how the GUI program is using Qt. Static analysis tools and system commands were used to analyze the application steps.

Through the static analysis of the STB application, the 70 classes of the Qt library are identified to be used and the modules including the unused classes are removed by applying the Qt configuration tool, qconfig. qconfig provides the ability to avoid compiling unnecessary sources to reduce the size of the Qt library for embedded systems. Based on the static analysis results, we constructed the optimized V1.0 Qt library using the qconfig tool.

The optimized library V1.0 using qconfig has the overall original files and the unnecessary parts of library sources are controlled by conditional macro statements to exclude the sources from compiling. Therefore, even if the Qt library has been optimized using qconfig, it will take a long time to build the executable file because all the original files should be compiled and the compiled files are included in the build image. From optimized library V1.0, we made the new optimized version V1.1 by physically removing the files that are not used in V1.0. Therefore, the unused files are not included in the build process.

For optimized version V2.0, we removed the unused folders and modified the corresponding build-related files. The folder removal process started with the top level folder src and removed the folders of unused sources using the results of static and dynamic analysis. The Qt build process first refers to the *.pro file. The *.pro file contains information about subfolders to refer to * .pri files in the build process. Therefore, all deleted folders should be deleted from the * .pro file.

For the version V3.0 or more, we have optimized the Qt library source by modifying at the source code level. In this process, the coverage data of source code was referenced. In some cases, we used the coverage data to disable unused features at the configuration panel of qconfig and found the unused files. The found files were removed manually. After the file is removed, additional work is required. This is because another file can contain the removed file using #include. If a file includes the removed file using #include, an error will occur during pre-processing of the build. For this reason, when you delete a file in the optimization process, we must modify the #include statement in the file that contains the deleted file. The optimized versions of V3.0, V3.1, V3.2, and V3.3 were acquired by performing folder-specific optimization for the entire folders of the optimized library version V2.0.

To optimize the DirectFB library, we used pmap to list the libraries used in the GUI application. The number of libraries used was 15 among the 31 libraries of DirectFB. The unused 16 libraries were removed and the 3.2 MB library image was reduced to 2.4 MB.

Table 4 shows the changes of the image sizes of optimized versions as optimization process is performed step by step. The original Qt library of 19.57 MB was optimized to be 7.26 MB in program image size. In the case of DirectFB, 3.2 MB was reduced to 2.4 MB by removing the library modules that did not be loaded in memory. In detail, libQtCore and libQtGui, which were the main optimization targets in the optimization process, were initially 15.6 MB and finally reduced to about 5.5 MB in V3.3.

TABLE IV
SOFTWARE SIZES OF EACH OPTIMIZATION VERSIONS (MB)

	Qt	**DirectFB**	**Total**
Initial	19,57	3.2	22.77
V1.0	10.26	3.2	13.46
V2.0	9.48	3.2	12.68
V3.0	9.25	3.2	12.45
V3.3	7.26	2.4	9.66

Table 5 shows the measured memory usage for each optimization version. This data is obtained by processing the results of the pmap command that is supported by the target system. VM (virtual memory) is the size allocated on the virtual memory and PM (physical memory) is the actual memory usage loaded into main memory when the program is executing. The optimal image size of the STB application is related to the PM size. The PM size represents the size of the memory actually needed when the application program is running. The PM size can be used to determine whether software can be further optimized or not. The sizes of PM and image of the optimized version V3.3 is 4.66 MB and 7.26 MB, respectively. This means the image size of V3.3 can be reduced more than 2 MB.

TABLE V
RUN-TIME MEMORY OF EACH OPTIMIZATION VERSION (MB)

	Qt		**DirectFB**	
	VM	**PM**	**VM**	**PM**
Initial	17.83	7.19	2.36	1.60
V1.0	10.00	5.17	2.36	1.60
V2.0	9.26	5.17	2.36	1.60
V3.0	9.00	5.06	2.36	1.60
V3.3	7.03	4.66	2.36	1.60

The library optimization proceeds in a top-down fashion. We first optimized the library at the module level, then removed unused class files and source folders that had become empty by file removal. Finally, we removed the unused files and folders through modifications of the source code. For further optimization, we should go into the intra-class level so that we optimize the methods of classes. This implies that we have to remove unused methods or variables of classes. However, the intra-class level optimization will consume too much time and be prone to incur errors in source code level because the difficulty of the intra-class level optimization may be similar to the source level programming. When the original library is updated by version up, the optimized version of the intra-class level is difficult to be updated accordance to the update of the original version due to its source level modification. That was why we avoided the intra-class optimization.

As mentioned previously, the target board did not support the gprof and gcov tools and we could not have the dynamic analysis data for STB application execution. To solve this problem, the GUI part of the STB application was ported to a Linux machine, and the dynamic analysis data of the limited STB application and the libraries were acquired and used for the optimization process.

We optimized the size of the Qt library from 19.57 MB to 7.26. In case of the DirectFB library, 3.2 MB was reduced to 2.4 MB by removing the library modules that did not be loaded in memory. Although the size of the library image has been greatly reduced through optimization, the size of run-time memory is reduced from 8.79 MB to 6.26 MB. This means that the library optimization has more impacts on memory for storing program image than on memory for program run.

IV. CONCLUSIONS

In this paper, we optimized the Qt and DirectFB library for an STB GUI application using a systematic top-down software optimization approach. Since the development environment of embedded systems are often error-prone and unstable, various obstacles may occur during optimization process. We cannot obtain the exact run-time information of library code if dynamic analysis tools does not work on a target system. To overcome this situation, we ported a shrunk version of the application program on a Linux PC and extracted partial dynamic analysis data. The partial data enables us to figure out the unused library parts and it is very useful for accelerating optimization.

Although the size of the library image has been greatly reduced through optimization, the size of run-time memory of the whole UI software is reduced by 12% from 61.1 MB to 53.9 MB while the size of run-time memory of Qt is reduced from 7.2 MB to 4.7 MB. This means that optimization of software beyond the Qt and DirectFB libraries is required for overall software optimization.

Software optimization is one of many desirable goals in software engineering and is often antagonistic to other important goals such as stability, maintainability, and portability. When an open source library is optimized for a product, it may not be easy to maintain the optimized library according to the update of the original library. However, if a developer is planning a line-up of embedded products, it may be advantageous to maintain an optimized library like a private code rather than continually optimizing large original library code for each product.

Almost all compilers support software optimization at the statement level, but few integrated tools are available to optimize at the module, file, and function levels. Future research on integrated library optimization tools that use static and dynamic analysis information is very important.

ACKNOWLEDGMENT

This work was supported by Hankuk University of Foreign Studies Research Fund of 2018.

REFERENCES

[1] S. Memeti, S. Pllana, A. Binotto, J. Kołodziej, and I. Brandic, "Using meta-heuristics and machine learning for software optimization of parallel computing systems: a systematic literature review," *Computing*(2018), https://doi.org/10.1007/s00607-018-0614-9, Springer Vienna.

[2] Andrizal, R. Chadry, and A. I. Suryani, "Embedded system using field programming gate array (FPGA) myRIO and LabVIEW programming to obtain data patern emission of car engine combustion categories," *International Journal of Informatics Visualization*, vol. 2, no. 2, pp. 56-62, 2018.

[3] S. Sushko and A. Chemeris, "The dependence of microprocessor system energy consumption on software optimization," in *Proceedings of 2017 IEEE 37th International Conference on Electronics and Nanotechnology*, pp. 451–454, Kiev, Ukraine, Apr. 2017.

[4] A. K. Sarma, "New trends and challenges in source code optimization," *International Journal of Computer Applications*, vol. 131, no. 16, pp. 27–32, Dec. 2015.

[5] M. C. M. Neto, S. S. Andrade, and R. L. Novais, "Cross-platform multimedia application development: for mobile, web, embedded and IoT with Qt/QML," in *Proceedings of the 23rd Brazillian Symposium on Multimedia and the Web*, pp. 23–26, Gramado, RS, Brazil, Oct. 2017.

[6] H. Zeng, Z. Zhang, and L. Shi, "Research and implementation of video codec based on FFmpeg," in *Proceedings of the 2016 International Conference on Network and Information Systems for Computers*, pp. 5-17, Wuhan, China, Apr. 2016.

[7] M. P. Mariano, "Application-level virtual memory for object-oriented systems," PhD Eng. thesis, Lille University of Science and Technology, France 2012.

[8] G. Polito, "Virtualization support for application runtime specialization and extension," PhD Eng. thesis, Lille University of Science and Technology, France 2015.

[9] K. A. Marianna, R. O. Lhoták, J. Dolby, and F. Tip, "Type-based call graph construction algorithms for Scala," *ACM Transactions on Software Engineering and Methodology*, vol. 25, no. 1, pp. 1–43, Dec. 2015.

[10] G.-H. Kang, Y. C. Kim, G. S. Yi, Y. S. Kim, Y. B. Park, and H. S. Son, "A practical study on code static analysis through open source based tool chains," *KIISE Transactions on Computing Practices*, vol. 21, no. 2, pp. 148–153, Feb. 2015.

[11] I. Giagkiozis, R. J. Lygoe, and P. J. Fleming, "Liger: an open source integrated optimization environment," in *Proceedings of the 15th Genetic and Evolutionary Computation Conference*, Amsterdam, The Netherlands, pp. 1089–1096, July 2013.

[12] S.-Y. Cho, "Library Optimization for Embedded Systems Using Open Tools," in *Proceedings of the 2018 International Conference on Electronics, Information and Communication*, pp. 674–675, Hawaii, USA, Jan. 2018.

[13] D. Grbić, M. Vranješ, B. Kovačević, and M. Milošević, "Hybrid electronic program guide application for digital TV receiver," in *Proceedings of 2017 IEEE 7th International Conference on Consumer Electronics – Berlin*, pp. 177–180, Berlin, Germany, Dec. 2017.
https://web.archive.org/web/20170603093935/http://directfb.net/

[14] T. Cséri, "Examining structural correctness of documentation comments in C++ programs," in *Proceedings of 2015 IEEE 13th International Scientific Conference on Informatics*, pp. 79–84, Poprad, Slovakia, Nov. 2015.

[15] Y. Zheng , S. Kell , L. Bulej, H. Sun, and W. Binder, "Comprehensive multiplatform dynamic program analysis for Java and Android," *IEEE Software* , vol. 33, Issue 4, pp. 55–63, Jul. 2016.

[16] pmap, [Online]. Available: https://linux.die.net/man/1/pmap

[17] M. Stevanovic, *Chap. 12 Linux Toolbox*, in Advanced C and C++ Compiling 1st Ed., Berkeley, CA: Apress, 2014.

[18] A.-D. Schmidt, R. Bye, H.-G. Schmidt, J. Clausen, O. Kiraz, K. A. Yuksel, S. A. Camtepe, and S. Albayrak, "Static analysis of executables for collaborative malware detection on Android," in *Proceedings on IEEE International Conference on Communications*, Dresden, Germany, Jun. 2009, https://doi.org/10.1109/ICC.2009.5199486, IEEE