Cslim: Automated Extraction of IoT Functionalities from Legacy C Codebases

Hyogi Sim[†,*], Arnab K. Paul[*], Eli Tilevich[*], Ali R. Butt[*], Muhammad Shahzad[‡]

Oak Ridge National Laboratory[†], Virginia Tech[*], North Carolina State University[‡]

simh@ornl.gov,{akpaul,tilevich,butta}@vt.edu,mshahza@ncsu.edu

ABSTRACT

Many Internet of Things (IoT) devices are resource-poor, possessing limited memory, disk space, and processor capacity. To accommodate such resource scarcity, IoT software cannot include any extraneous functionalities not used in operating the underlying device. Although legacy systems software contains numerous functionalities that can be reused in IoT applications, these functionalities are exposed as part of a larger codebase with multiple complex dependencies and a heavy runtime footprint. To enable programmers to effectively reuse extant systems software in IoT applications, this paper presents Cslim, a cross-package function extraction tool for C. Cslim extracts programmer-specified functions from a source package and generates new source files for a target package, thereby enabling the reuse of systems software in resource-poor execution environments, such as the IoT devices. Cslim resolves all dependencies by recursively extracting required functions, while bypassing the complexities of preprocessor macro variabilities by operating on preprocessed source files. Furthermore, Cslim efficiently traverses and resolves the calling dependencies by maintaining an in-memory relational database. Finally, Cslim is easy to use, as it requires neither manual intervention nor source code modifications. Our prototype implementation of Cslim has successfully extracted a set of functions from SQLite and GlusterFS, producing slimmed down executables that can be deployed on IoT devices.

ICDCN '19, Bangalore, India

DOI: 10.1145/3288599.3296013

CCS CONCEPTS

•**Software and its engineering → Embedded software;** *Maintaining software;* Software usability;

KEYWORDS

Software Engineering, IoT

ACM Reference format:
Hyogi Sim[†,*], Arnab K. Paul[*], Eli Tilevich[*], Ali R. Butt[*], Muhammad Shahzad[‡]. 2019. Cslim: Automated Extraction of IoT Functionalities from Legacy C Codebases. In *Proceedings of International Conference on Distributed Computing and Networking, Bangalore, India, January 4–7, 2019 (ICDCN '19),* 6 pages.
DOI: 10.1145/3288599.3296013

1 INTRODUCTION

The C language is the lingua franca of systems software. This language naturally fits the implementation requirements of various low-level system components, such as operating systems and firmware, which put an emphasis on increasing execution efficiency and reducing runtime footprint. C programs can be compiled into a minimal number of machine instructions that can be deployed in compact binaries. In particular, minimizing the executable binary code size becomes crucial in restricted hardware and software environments (e.g., memory and storage capacity, available shared libraries, etc.). Notably, an emerging trend of Internet of Things (or IoT) [14] envisions intelligent and connected physical objects, from small hand-held devices to vehicles and large buildings. Undoubtedly, the binary code size can negatively impact the runtime performance, power usage, and building cost of small-scale IoT devices.

One of the peculiarities of C programming is a lack of standard libraries that represent common data containers and algorithms to facilitate the development process [6]. In C, even a string is simply a `null` terminated array of characters that must be explicitly allocated. Higher-level libraries, such as `glib` [5], and `qt` [8], introduce a space deployment overhead, as they require the shipment of an entire shared library to be able to use only a single module, such as a hash table; this overhead can be prohibitive for deployments in small or restricted environments. To eliminate the overhead, programmers

manually extract functions of interest from other software packages, and modify them accordingly to fit a target software package. Oftentimes, however, such desired functions are chained in a complex calling dependency in the package, rendering manual extraction tasks tedious and error-prone. Software refactoring—behavior-preserving code transformations [19]—can automatically extract these desired functions. Refactoring can address the needs of the IoT community by reusing the already stable and tested infrastructure to build applications for IoT devices.

In this paper, we present CSLIM, a cross-package function extraction tool for C. As an input, programmers only need to specify a list of functions to be extracted from the source package. Once the function list is provided, CSLIM first scans the source package, analyzes the calling dependencies, and creates a reference database. It then recursively resolves the calling dependencies, and calculates the final list of functions in the correct order to appear in the new source files. Finally, CSLIM generates new `.h` and `.c` files, that are self-contained and thus can be embedded in other software packages. CSLIM sidesteps the complexity of handling C preprocessor macros by operating on the output files of the C preprocessor. As stated above, CSLIM targets restricted environments, such as IoT devices. Consequently, injecting static package configurations eliminates variabilities, without compromising the efficacy and practicability of CSLIM. Furthermore, CSLIM only manipulates the package source code, thus being architecture independent.

To evaluate our prototype of CSLIM, we used `SQLite` [9] as a test case because of its popularity in Android application development. `SQLite` is preferred in small devices, due to its lightweight nature and single-tier database architecture. `SQLite` was designed to provide local data storage for individual devices and applications. Therefore, `SQLite` databases require little administration, making them particularly well-suitable for devices that need to operate without expert human support, such as those used in the "internet of things." In addition to `SQLite`, we also evaluate CSLIM by successfully extracting functions from `GlusterFS` [10], another open-source C package. `GlusterFS` is a scalable, distributed file system that is well-suited for data-intensive tasks, such as media streaming and cloud storage.

The remainder of the paper is organized as follows. We explain our design (§ 2), and implementation (§ 3) of CSLIM, followed by our initial experience and findings from our prototype (§ 4). We also present related research work (§ 6), and then our conclusion and some future directions (§ 7).

2 DESIGN OF CSLIM

CSLIM has been designed to conform to the following criteria, to ensure its practicality.

No manual source code modification. Requiring modifications to existing source may hurt ongoing development productivity and code maturity. Moreover, requiring source code modification prevents the inclusion of new source packages, decreasing the tool's adaptability.

No manual processing. Semi-automated refactoring can unreasonably burden developers, particularly for larger codebases. Therefore, it is essential to obviate the need for human intervention to ensure practicality.

Ease of maintenance. Individual source packages can always be updated (e.g., to fix bugs, to add new features, etc.) after the needed functions have been extracted. To accommodate such updates, the framework should support incremental updates that free programmers from hand-operated and error-prone manual patching.

Figure 1 shows the overview of CSLIM. The user specifies the list of functions to extract from the source package, which we refer to as *target functions*. CSLIM first bootstraps the source package, primarily to avoid complications of preprocessor macro variabilities. After the bootstrapping, the source files in the source package are scanned and all calling dependencies between functions are analyzed. CSLIM stores the analysis results in the *reference database*. Next, any calling dependencies in target functions are resolved by consulting the reference database. Finally, CSLIM generates the self-contained target source files ready to be embedded in the target package. We next detail each step of CSLIM in turn.

2.1 Bootstrapping the source package

To start extracting functions from a source package, we first bootstrap the source software package in two phases: *configuring the source package* and *running the C preprocessor*. In most cases, configuring a source package requires running its `configure` script, which takes input flags that specify the target device architecture and other building options, and generates appropriate build scripts. The output build scripts contain all necessary information to resolve the variabilities of preprocessor macros. Then, the bootstrapping is completed by running the build scripts (e.g., `Makefile`), but only up to the C preprocessor, as CSLIM works at the source code level. The output source files, with all preprocessor macros stripped, can be further processed by CSLIM.

The preprocessor macros are known to significantly complicate the analysis and transformation of C programs [12, 13, 20]. However, our bootstrapping step eliminates the need for CSLIM to directly address these

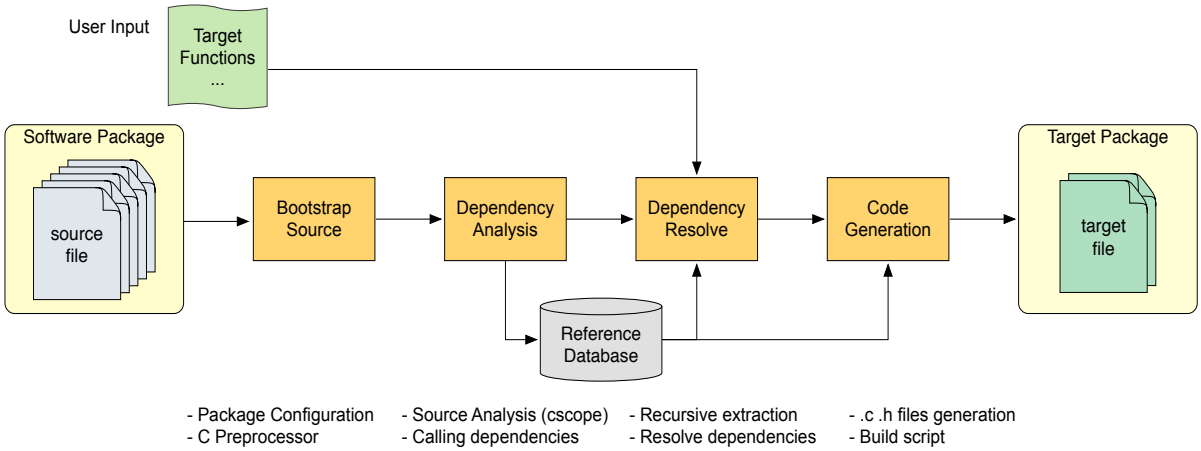

Figure 1: Overview of the code extraction framework in CSLIM. Based on the target function list, CSLIM analyzes the source package, resolves the calling dependencies in target functions, and generates the target source files.

complications. This shortcut should not affect the practical use of CSLIM, given our target development environments, which possess limited computing resources and scarce libraries, (e.g., IoT devices) in which most software packages are statically configured and optimized specifically for a single target architecture.

2.2 Analyzing the source code

After the bootstrapping, CSLIM analyzes the source code to identify any possible calling dependencies within the functions to extract. For a systematic analysis, CSLIM records all functions and their calling interactions in a relational database, whose data schema appears in Figure 2. The Functions table records all functions in the source package, each with its name and definition's location, including the source file name and the line number. The Calls table records all calling occurrences in the source package. For instance, in Figure 2, function `caller` calling function `callee` means that if `caller` is extracted, `callee` should be extracted as well. CSLIM uses this reference database to resolve any calling dependencies between functions in the following steps.

Functions

FID	Function Name	Source File	Line Number
1	caller	caller.c	24
2	callee	callee.c	51
...	...	...	...

Calls

CID	Caller FID	Callee FID
1	1	2
...	...	...

Figure 2: The simplified database schema to describe the function interactions in the source package. This example shows the table status of recording the calling relationship between `caller` and `callee`, in which `caller` calls `callee`.

CSLIM needs to scan all the files in the source package to populate the reference database. CSLIM exploits traditional UNIX tools, `cscope` [3] and `ctags` [2], for scanning, as further described in § 3.

2.3 Resolving the calling dependencies

After the source package has been analyzed, CSLIM recursively resolves any dependencies in the target function list, as presented in Listing 1.

First, both `targets` and `unresolved` lists are populated with the target functions, via `populate_from_input()` procedure. While `targets` is a static data structure, `unresolved` keeps changing during the resolving process. For instance, a function is initially added to the `unresolved` list, and then removed as soon as its calling dependencies are resolved. The algorithm resolves dependencies by adding extra functions to be extracted into the `compats` list. Specifically, the `while` loop inspects the functions in the `unresolved` list, which is initially populated with the target functions. For each function (`f`) in the `unresolved` list, the algorithm queries, via invoking `get_callees()` to acquire any functions (`callees`) called by the function `f`. If there exists any, such `callees` are added to both `compats` and `unresolve` lists. Then, the function `f` can be removed from the `unresolved` list. In this manner, the loop continues until

```
unresolved = []
targets = []
compats = []

def populate_from_input():
  # read input file and return as a list
  ...

def get_callees(f):
  # look up callees from the reference
  # database
  ...

targets = populate_from_input()
unresolved = populate_from_input()

while unresolved.len() > 0:
  for f in unresolved:
    callees = get_callees(f)
    for c in callees:
      if c in not targets or compats:
        unresolved.append(c)
      unresolved.remove(c)
      compats.append(c)
```

Listing 1: The pseudo code of recursively resolving calling dependencies among target functions in CSLIM.

the `unresolved` list becomes empty. Upon the completion, the `target` list holds the target functions (in the appearing order from the user input), while the `compat` list contains extra functions in the order the calling dependencies are resolved. As we will see shortly (§ 2.4), keeping the order or the `compat` is important in generating the correct source file.

2.4 Generating the source code

First CSLIM generates the source files that can be included in the target package. In addition to the source files with the target functions, e.g., `functions.h` and `functions.c`, CSLIM also generates `compat.h` and `compat.c`, containing the declarations and definitions, respectively, of the extra functions extracted to satisfy the calling dependencies of the target functions. All source files are generated in a target directory, e.g., `/project/compat`, which can be specified by the user.

First, the target functions are extracted in the order of the `targets` list in Listing 1, which exactly follows the order of the target function list, provided by the user. For each function, CSLIM extracts the function signature (or prototype) and definition from the original source files, by consulting the *reference database* that was previously built (§ 2.2). Next, the functions in the `compats` list in Listing 1, are extracted and written to the resulting source files, `compat.h` and `compat.c` in the reverse order. Note that the appearing order in the `compat.c` is important to resolve the calling dependencies between those functions. For instance, if function *A* is called by another function *B*, function *A* should be written before function *B* to avoid any potential compile errors.

After the source files are generated, CSLIM finally generates a build script, `Makefile.am`. The new build script can facilitate the manual integration process, particularly when a package adopts the popular `autoconf`-based build system.

3 IMPLEMENTATION

We prototyped CSLIM with four separate programs written in C++ and Python. For the initial analysis of the source package, CSLIM relies on the output of `cscope` and `ctags`, which are readily available in most UNIX-like systems. Specifically, `cslim-createdb` program, written in C++, parses the output files of `cscope` and `ctags` and creates the reference database using SQLite [9]. `cscope` is a popular development tool for browsing C code. `ctags` works by generating an index or (a tag) file of the program names found in the input source and header files. This tag file allows definitions to be quickly and easily located by any utility tools.

The remaining steps (i.e., resolving dependencies (§ 2.3) and generating source codes (§ 2.4)) are implemented in Python. An additional SQLite database file maintains data structures while resolving the dependencies. Finally, a bash script, `cslim.sh` and a `Makefile` automate all extraction steps. The bash script takes the name of the input file, which lists target functions to be extracted, as its argument.

4 EMPIRICAL STUDY

In this section, we demonstrate the viability of CSLIM by discussing experiments with two real-world softwares written in C: SQLite [9] and GlusterFS [10].

4.1 Synthetic C Software

We wrote a synthetic C program that has internal calling dependencies, to verify the correctness of CSLIM. Specifically, the program consists of a single header file and four C source files. Each C file has 10 functions (40 functions in total), calling each other 37 times. We also wrote a main function, which calls 6 of the total 40 functions. In the baseline case, we compile all C files to generate a single executable file. Using CSLIM, we extract the functions from the synthetic C files and compile the program with the output files generated by CSLIM. Table 1 compares the result.

The number of functions are accounted from the resulting executable file with `nm` command [4, 7], particularly by counting the "T" section (text/code section). In

	Number of functions	Code size
Baseline	46	13,336 Bytes
CSLIM	20	8,432 Bytes

Table 1: Comparison of resulting binary executables.

the baseline case, all 40 functions from the source package are included in the executable file. The executable file also includes our main function and 5 other code sections that are inserted by the compiler (e.g., `_init`, `_fini`, `_start`). In comparison, CSLIM generated the executable file only with 20 functions. In fact, the calling dependencies (from the 6 target functions) require 8 more functions to be extracted, resulting in extracting 14 functions. Note that exactly 20 functions appear in the resulting executable file, including the main function and 5 compiler-generated functions. The size of the executable file also effectively decreases, only 63% of the baseline executable file.

4.2 Real-World C Software

We have also tested CSLIM to extract functions from SQLite [9] and GlusterFS [10]. Specifically, we have extracted `sqlite3PageMalloc` and `sqlite3PageFree` functions from SQLite, and all functions from the dictionary implementation (`dict_*`) in GlusterFS. A total of 14 and 103 extra functions have been additionally extracted respectively from SQLite and GlusterFS, to satisfy the calling dependencies.

5 DISCUSSION AND FUTURE WORK

We discuss the limitations that we have observed from conducting the experiments. Although CSLIM can extract functions as expected, the output files still require few manual tasks for the following reasons. First, CSLIM currently identifies the function body based on simple string matching, which only works with some coding styles. Second, our current prototype cannot correctly extract custom type definitions that are required by the target functions. These limitations could be addressed by replacing the current parser with a more advanced C source code parser, such as SuperC [13]. Also, CSLIM cannot autonomously modify the build script to stop the build process after the C preprocessor, e.g., `gcc -E`. Therefore, the user has to perform the bootstrapping process (§ 2.1) by hand, before executing CSLIM. We plan to address these limitations in the future.

In addition, we plan to integrate the extra features into CSLIM for providing a better usability. First, we plan to append an additional step after the code generation, to verify that the correctness of the extracted functions is not compromised. Lastly, we also plan to add the ability to incrementally merge any source updates (e.g., bug fixes) into the target source code by integrating with a version control system (e.g., `git`).

6 RELATED WORK

We discuss the general challenges of refactoring C programs, along with the solutions offered by prior research works. The biggest challenge in refactoring C programs is the handling of C preprocessor macros, which is often treated as a separate language [1, 11]. In essence, a macro is a name given to a block of C statements as a pre-processor directive. The `#define` directive defines an identifier and a character sequence that will be substituted for the identifier each time it is encountered in the source file. The identifier is referred to as *a macro name* and the replacement process as *macro replacement*.

TypeChef [16] detects type errors without having to generate all variants imposed by `#ifdef` directives. In TypeChef, the preprocessor partially processes the preprocessor directives, and separately analyzes the variabilities. Although the approach of TypeChef is promising, its parser requires implementation-specific knowledge about the target source code and cannot analyze arbitrary C codes. Another approach of handling C preprocessor macro is to design and develop a new preprocessor language for C. For instance, ASTEC [17] features similar functionality and usage to the original C preprocessor macro, but is analyzable and refactorable. However, it lacks support for C++. Moreover, converting existing source code to ASTEC framework requires human intervention, making it practically less appealing. Yacfe [21] is a parser that can analyze C/C++ source code without preprocessor macros. This heuristics-based parser can handle large C projects, at the cost of compromising correctness. A similar heuristics-based approach is supported by Coccinelle [22], which automatically generates documentation and collateral evolutions as a patch file to assist developers of device drivers in the Linux kernel. SuperC [13] is the first framework that can completely parse C programs including the preprocessor macros. The preprocessor of SuperC preserves the variability, which is subsequently resolved by the parser that spawns multiple processes for each static condition. Despite these alternatives, experienced developers still often resort to using the C preprocessor for portability and variability reasons [18]. Although refactoring the preprocessed C code is ill-advised in general [12], CSLIM leverages the unique features of our target environment (i.e., IoT devices), thus safely eliminating the source code variabilities and operating on the preprocessed code.

There are many works which focus on optimizing resource consumption in distributed systems [23, 24].

In this paper, we focus on several works which optimize consumption of resources in constrained environments for other languages. For instance, a code cache management technique dynamically unloads dead and infrequently used code in JIT-based JVMs for mobile and embedded devices [26]. Specifically, the presented system profiles applications online and offline to select candidates for code unloading. Other techniques have also been used to reduce memory consumption in such environments. To balance memory and performance by reducing re-translation, selective flushing of software code has been proposed [15]. In addition, managing code cache while maintaining correctness and minimizing memory consumption has also been proposed for resource constrained environments [25]. CSLIM compliments the aforementioned techniques, as it reduces the code size in such resource constrained environments by selectively extracting a minimal amount of target functions from generic C libraries.

7 CONCLUSION

We presented CSLIM, an automatic source-code refactoring tool that extracts functions of interest from a source package after resolving their calling dependencies, while bypassing the preprocessor macro complexities, and then generates source code that can be directly included in any target packages. CSLIM supports restricted development environments, such as IoT devices, vastly reducing their dependence on external library code. Without requiring any source code modifications or manual intervention, CSLIM can be easily adopted as a development tool by IoT developers.

REFERENCES

[1] 1987. The C Preprocessor - GCC - GNU. https://gcc.gnu.org/onlinedocs/cpp/. (1987). Accessed: Sept, 2018.

[2] 2009. Exuberant Ctags. https://ctags.sourceforge.net/. (2009). Accessed: May, 2018.

[3] 2012. Cscope Home Page. https://cscope.sourceforge.net/. (2012). Accessed: July, 2018.

[4] 2016. Autoconf - GNU Project - Free Software Foundation. http://www.gnu.org/software/autoconf/autoconf.html. (2016). Accessed: Sept, 2018.

[5] 2018. GLib - GNOME Developer Center. https://developer.gnome.org/glib/stable/. (2018). Accessed: July, 2018.

[6] 2018. Glibc - GNU. https://www.gnu.org/software/libc/. (2018). Accessed: May, 2018.

[7] 2018. GNU Binutils. https://www.gnu.org/software/binutils/. (2018). Accessed: Sept, 2018.

[8] 2018. Qt - Home. https://www.qt.io/. (2018). Accessed: Sept, 2018.

[9] 2018. SQLite Home Page. https://www.sqlite.org/. (2018). Accessed: Sept, 2018.

[10] 2018. Storage for your Cloud. — Gluster. http://www.gluster.org. (2018). Accessed: Sept, 2018.

[11] Alejandra Garrido and Ralph Johnson. 2002. Challenges of Refactoring C Programs. In *Proceedings of the International Workshop on Principles of Software Evolution (IWPSE '02)*.

[12] A. Garrido and R. Johnson. 2003. Refactoring C with Conditional Compilation. In *2003. Proceedings. 18th IEEE International Conference on Automated Software Engineering.*

[13] Paul Gazzillo and Robert Grimm. 2012. SuperC: Parsing All of C by Taming the Preprocessor. *SIGPLAN Not.* 47, 6 (June 2012).

[14] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. 2013. Internet of Things (IoT): A Vision, Architectural Elements, and Future Directions. *Future Generation Computer Systems* 29, 7 (2013).

[15] Apala Guha, Kim Hazelwood, and Mary Soffa. 2010. Balancing Memory and Performance through Selective Flushing of Software Code Caches. In *Proceedings of the 2010 international conference on Compilers, architectures and synthesis for embedded systems.* ACM, 1–10.

[16] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. 2011. Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation. In *Proceedings of OOPSLA '11.*

[17] Bill McCloskey and Eric Brewer. 2005. ASTEC: A New Approach to Refactoring C. *SIGSOFT Softw. Eng. Notes* 30, 5 (Sept. 2005).

[18] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Sarah Nadi, and Rohit Gheyi. 2015. The Love/Hate Relationship with the C Preprocessor: An Interview Study. In *29th European Conference on Object-Oriented Programming (ECOOP 2015)*, John Tang Boyland (Ed.), Vol. 37.

[19] Tom Mens and Tom Tourwé. 2004. A Survey of Software Refactoring. *IEEE Transactions on Software Engineering* 30, 2 (2004).

[20] Jeffrey L. Overbey, Farnaz Behrang, and Munawar Hafiz. 2014. A Foundation for Refactoring C with Macros. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014).*

[21] Yoann Padioleau. 2009. Parsing C/C++ Code Without Preprocessing. In *Proceedings of the 18th International Conference on Compiler Construction: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009 (CC '09).*

[22] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. 2008. Documenting and Automating Collateral Evolutions in Linux Device Drivers. In *ACM SIGOPS Operating Systems Review*, Vol. 42.

[23] Arnab K Paul, Arpit Goyal, Feiyi Wang, Sarp Oral, Ali R Butt, Michael J Brim, and Sangeetha B Srinivasa. 2017. I/o load balancing for big data hpc applications. In *2017 IEEE International Conference on Big Data (Big Data).* IEEE, 233–242.

[24] Arnab Kumar Paul, Wenjie Zhuang, Luna Xu, Min Li, M Mustafa Rafique, and Ali R Butt. 2016. Chopper: Optimizing data partitioning for in-memory data analytics frameworks. In *2016 IEEE International Conference on Cluster Computing (CLUSTER).* IEEE, 110–119.

[25] Forrest J Robinson, Michael R Jantz, and Prasad A Kulkarni. 2016. Code Cache Management in Managed Language VMs to Reduce Memory Consumption for Embedded Systems. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 11–20.

[26] Lingli Zhang and Chandra Krintz. 2004. Profile-Driven Code Unloading for Resource-Constrained JVMs. In *Proceedings of the 3rd international symposium on Principles and practice of programming in Java.* Trinity College Dublin, 83–90.