Modeling, Analyzing and Optimizing Android System as Complex Networks

Die Li, Yuan Dong*, Huan Luo, Tingyu Jiang, Shengyuan Wang, Yu Chen
Department of Computer Science and Technology
Tsinghua University, Beijing 100084, China
*Email: dongyuan@tsinghua.edu.cn

***Abstract*—Due to the rise of mobile devices, there is an increasing interest in the optimizations of the mobile operating system both for performance and code size. However, the complexity of the mobile operating system makes these optimizations more challenging.**

This paper presents extended call graph (ECG) for modeling the access relations of functions and data objects of all the native code of Android except those in the Linux kernel. We identified that the ECG of Android is a typical complex network. It exhibits the basic features of scale-free topology and small-world structure. Based on these results, this paper found and eliminated the "isolated" vertices and subgraphs (without indegree and outdegree) in the graph to build a better connected and centralized graph. The largest connected subgraph before optimization contains only 87.0% of all the vertices while the subgraph after optimization contains 99.9% of all the vertices. Thus, most of the dead code in libraries and framework are eliminated automatically.

This method is applied to Android-x86 on ASUS Eee PC. It achieves about 26.7% code size reduction in total and up to 1.3% speedup in floating-point computation. It also can be adopted to model, analyze and optimize other mobile operating systems developed in C/C++ and assembly languages. This work not only provides a foundation of optimizing Android operating system for both performance and code size, but also helps us to understand and develop the complex software system more efficiently.

I. Introduction

With the rise of mobile devices, such as smartphones and pads, there is an increasing need to optimize the mobile operating system both for performance and code size. However, the complexity of the modern operating system like Android[1] makes these optimizations more challenging. Android is the most popular operating system for mobile devices, accounting for about 50% of the smartphone market share in the 4th quarter of 2011[2].

Android is composed of a number of complex software systems. As shown in Figure 1, it adopts a layered architecture[3]. All applications are written in the Java programming language (sometimes with Java native interface, i.e. JNI[4], to call native code). The application framework includes both Java and C/C++ code. It provides services, such as activity manager and window manager, for applications. The libraries and Android runtime are mainly implemented in C/C++. Underneath all of the above is the Linux kernel.

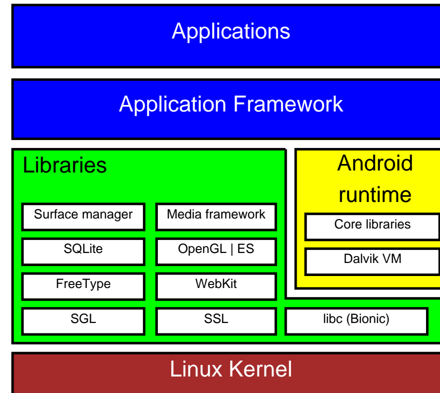

Fig. 1. Android architecture.

TABLE I
Code line statistics of Android-x86 3.2.2 (in kilo, generated by SLOCCount[5]).

	C/C++	Asm	Java	Others	Total
Applications	182	0	525	1	708
App framework	783	7	638	4	1,432
Android runtime	282	53	538	7	880
Libraries	8,073	127	568	467	9,234
Linux kernel	9,549	241	0	27	9,817
Others	346	0.1	257	22	626
Total	19,215	428	2,526	528	22,697

The total physical source lines of code of Android-x86 3.2.2 (version 20120215) is 22,697k. It is developed in Java, C/C++ and other programming languages. Table I shows the code line statistics of different components.

The system contains a lot of dead code that are completely unreachable, because the source code come from different developers, and some from open source communities whose main platforms are not mobile systems. Android uses its own C library, called "Bionic",

composed partly from the BSD C library combined with Android original code[6]. On the x86 architecture, the size of a stripped Bionic library is only 26% of that of a stripped GNU C Library[7]. It achieves the small code size by not providing unnecessary interfaces. For example, many inter-procedural communication interfaces are not implemented because Android has its own "intent" service[6].

Therefore, it would be valuable to eliminate the dead code from all the native code of Android except those in the Linux kernel (There are about 9,507k lines of C/C++ and assembly, as shown in the shadowed cells in Table I). But there are two fundamental challenges that must be addressed first.

Challenge #1: How to model the complex Android system? With a sound theoretical basis and a variety of applications, complex networks offer a perfect nonlinear abstraction when analyzing applications to real-world problems[8]. Extensive research[9], [10] has confirmed that class collaboration graphs and static call graphs of a large scale software such as eMule, Openvrml, MySQL, and Linux Kernel display scale-free topology and small-world properties of complex networks, which have also been found in networks built with inter-package dependency in Linux distributions[11], [12]. Most of the researches on network of open-source software focus on single software or on the package-level relations. None of them handles function-level relations in multilingual complex systems as Android.

Challenge #2: How to identify and eliminate the dead code in Android? Dead code is common in real-world software[13]. Most modern compilers build a control flow graph to eliminate dead code in a function. Some compilers perform a similar operation on the compilation-unit level to remove functions and objects that are local to a compilation unit but never used. More sophisticated compilers can perform inter-procedural or link-time optimizations[14][15][16] to remove unused functions and objects on the program level, which usually involves a call graph for the whole program. However, these existing methods have no ability to solve the problem we face here.

Our Contributions. This paper proposes an optimization method based on modeling and analyzing the Android system as complex networks. Based upon the previous work in open source software analysis and link-time optimization, this paper makes the following contributions:

- Our work presents extended call graph (ECG) for modeling the relations of functions and data objects in relocatable files[1]. Then we build the extended call graph of all the native code (C/C++/Asm) of Android except those in the Linux kernel.
- We identified that the ECG of Android is a typical complex network. It exhibits the basic features of scale-free topology and small-world structure. We also found that there are many "isolated" vertices and subgraphs (without indegree and outdegree) in the graph. It means that there are plenty of vertices that have no interactions with the rest of the graph.
- Based on these results, this paper eliminates the "isolated" vertices and to build a better connected graph. The original graph contains only 87.0% of all the vertices while the graph after optimization contains 99.9% of all the vertices. Thus, most of the dead functions and inaccessible variables in libraries and framework are eliminated automatically.

To our best knowledge, this is the first work to construct and analyze the complete extended call graph for functions and data objects of Android system, and then refine the graph to perform system-wide dead code elimination. This method is applied to Android-x86 3.2.2. It achieves about 26.7% code size reduction in total and up to 1.3% speedup in floating-point computation.

The rest of this paper is organized as follows: we first present an informal overview of our approach to analyze and optimize Android system (Sec II). Then we define, construct and analyze the extended call graph of Android system as complex networks (Sec III). Based on these results, we present a system-wide binary rewrite optimization to eliminate dead code (Sec IV), and show the evaluation of the real system (Sec V). Finally we discuss related works (Sec VI), draw conclusions and discuss future work (Sec VII).

II. Overview

Figure 2 outlines the steps of our analysis and optimizations. In the default build process of a system with multiple binaries, each binary (executable or shared library) is compiled and linked separately, and the compiler and linker only use information from the source files for one binary. In a complex system such as Android, binaries produced in this manner are usually not optimal. For example, a shared library may contain externally visible symbols that are never actually used.

In our method, we construct a whole-system extended call graph (ECG) by extracting information from relocatable object files. The ECG is a directed graph.

[1] Relocatable files are commonly called "object files." Because we directly manipulate ELF files, we follow the ELF specification[17] to use "relocatable files" or "relocatable object files", while "object files" includes relocatable, executable and shared object files.

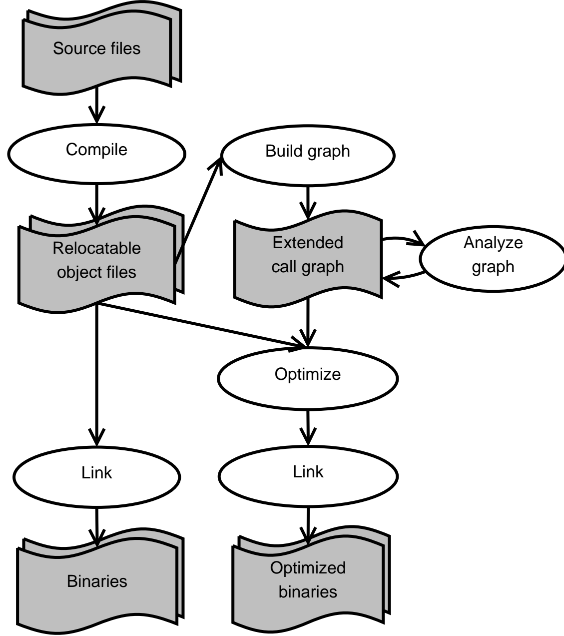

Fig. 2. Overview of our approach (right). Compare with the default build process (left).

Each vertex is a section in a relocatable object file, containing one or more functions or data objects. An edge in the graph represents a "use" of the successor by the predecessor. There are also other uses of a section the edges cannot represent. For example, the startup code of an executable is called by the OS, and some functions can be called by Java code through JNI. Therefore, we also include a set of entry points, i.e. sections that might be used in a manner not represented by any edge. A more formal definition of the ECG is given in Sec III-A.

The ECG exhibits scale-free (SF) topology[18] and small-world (SW) structure[19]. The former states that the proportion of vertex $P(k)$ which have k degrees decays as a power law $P(k) \approx C * k^{-\gamma}$. The latter refers to the fact that network tends to have small average path length between vertices along with a large clustering. We also use PageRank[20][21] to evaluate the relative importance of vertices. The results suggest a correlation between the PageRank and degree of a vertex. Most functions with the greatest PageRank and degrees are related to dynamic memory operations, which are frequently used in C and C++. Furthermore, the ECG consists of a great number of connected components, the largest of which contains more than 87% of all vertices while no other component contains more than 10.

From the results of the complex networks analysis, we conjecture that substantial parts of the Android code are actually unreachable and never used. In order to identify them more accurately, we define the significant subgraph of the ECG, which consists of vertices that are actually reachable and must be retained. Vertices not in the significant subgraph can be eliminated from the final binaries. The significant subgraph can be found with a straightforward graph search algorithm. We rewrite the relocatable files to mark sections not in the significant subgraph in a way that enables the linker to discard them at link time.

Finally, we evaluate the size, correctness and performance of the optimized system and compare them with those of the original system. The evaluation suggests a substantial reduction in size, unchanged correctness, and improved performance.

III. Android as Complex Networks

A. *Definition of Extended Call Graph*

The Extended Call Graph is defined in terms of relocatable object files. Relocatable object files are language independent and simpler than high-level languages. They also retain more useful information than final binaries. Each relocatable object file is composed of a number of sections. Each section contains one or more entities.

Definition 1. A *data object* is an object statically allocated in a data, rodata, bss or similar section. An *entity* is either a function or a data object.

In C/C++, data objects include all global and local static variables and objects, including compiler-generated objects such as virtual function tables.

A relocatable object file can be denoted by the set of sections in it, and a section by the set of entities in it, e.g. $R = \{S_1, S_2\}$ denotes a relocatable object file containing sections S_1 and S_2, and $S_1 = \{e_1, e_2\}$ denotes a section containing entities e_1 and e_2.

Definition 2. Section S is a *direct user* of section S', or S *directly uses* S', if there is at least one entry relative to S' or any $e' \in S'$ in the relocation data for S.

From the source-level perspective, S is a direct user of S' if one of the following conditions holds:

- A function $e \in S$ directly calls a function $e' \in S'$ or assigns the address of e' to a pointer. A function call is *direct* if the the name of the callee is known at compile time, as opposed to a call through a function pointer whose value varies depending on run-time context. Therefore, a call made through the procedure linkage table (PLT) is direct though a function pointer is involved in implementation details, while a function call through a virtual function table is not direct.

- A function $e \in S$ reads, writes, or takes the address of, data object $e' \in S'$.
- A data object $e \in S$ or, if e is of an aggregate type, one of its members, is initialized by the address of $e' \in S'$, optionally plus a constant offset.[2]

Definition 3. A section S is an *entry point* if any of the following conditions holds:

1) Section S contains the startup code of a process;
2) There exists an entity $e \in S$ that may be used through dynamic binding at run time;
3) Section S is not a regular code or data section (text, data, rodata, bss).

An entry point is a section which can be used at run time, but such use is not reflected by the "direct use" relation in Definition 2. Section III-B provides a more detailed description of entry points.

Definition 4. The *extended call graph* for a system (compiled to relocatable object files) is a directed graph $G = (V, E, R)$, where vertex set V is the set of sections in relocatable object files, edge set $E = \{(S, S') \in V \times V : S \text{ directly uses } S'\}$, and $R \subseteq V$ is the set of entry points.

Normally, the compiler places entities defined in the same compilation unit and of the same type in one section. This default behavior would render the ECG too coarse. Fortunately, many modern compilers have a feature to compile each function and data object into a separate section. Hereinafter it is assumed the feature is always enabled, and each section contains exactly one entity, with few exceptions.[3] Sometimes a single-entity section is not strictly distinguished from the entity in it.

B. Construction of Extended Call Graph

We first define some related denotations.

Definition 5. Given entity e, $Sym(e)$ denotes the set of all its names (including aliases). Given section S, $Sym(S) = \bigcup_{e \in S} Sym(e)$ denotes the set of names of all entities in it.

Definition 5 helps handle aliases. In relocatable object files, all names of an entity are equally entered in the symbol table, without distinction between a "real name" and an alias. For example, if function (denoted by e) `foo` has an alias `bar`, the section (denoted by S) for the function may be named `.text.foo` or `.text.bar`, and $S = \{e\}$, $Sym(S) = Sym(e) = \{\texttt{"foo"}, \texttt{"bar"}\}$.

Definition 6. Given symbol name s, $Sec(s)$ denotes the set of sections which has an entity named s, i.e. $Sec(s) = \{S : s \in Sym(S)\}$.

Definition 7. Given section S, $SecUse(S)$ denotes the set of sections in the same relocatable object file which are directly used by S, $SymUse(S)$ denotes the set of symbol names which are referenced in the relocatable data of S but cannot be resolved in the same relocatable object file, and $Use(S)$ denotes the set of sections in the whole system that are directly used by S.

When a section S references a symbol name s, i.e. $s \in SymUse(S)$, the actually used section is in $Sec(s)$. It is possible that section S may use different sections in $Sec(s)$ in different contexts. (For example, the process startup code calls different main functions when linked into different executables.) Therefore we have the following conclusion:

Theorem 1. For a section S, $Use(S) \subseteq Use'(S)$, where

$$Use'(S) = SecUse(S) \cup \bigcup_{s \in SymUse(S)} Sec(s).$$

The steps to construct the ECG are as follows:

Step 1. Compile Android with additional compiler options so that each entity is compiled into a separate section in the relocatable object file.

Step 2. Read the relocatable object files and collect the following data: V: the set of sections in all relocatable object files; $Sym(e)$ for each entity $e \in S, \forall S \in V$; $Sym(S)$, $SecUse(S)$ and $SymUse(S)$ for each section $S \in V$. See Definitions 5 and 7.

With these data, $Use'(S)$ can be computed for each section S. See Definitions 6, 7 and Theorem 1.

Step 3. Merge data from Step 2 to get the set of edges: $E = \{(S, S') \in V \times V : S' \in Use(S)\}$.

However, it is a challenge to compute $Use(S)$ accurately for some sections, where a superset of $Use(S)$ can be used as a conservative fallback. We first compute $Use'(S)$ (see Theorem 1), and then try to remove some elements which are not in $Use(S)$ according to other information (such as strong and symbol symbols). As a result, the computed set E may contain slightly more edges than ideal. The percentage of such false edges is estimated to be below 0.5% in the ECG for Android.

Step 4. Get the set R of entry points. In consistence with Definition 3, identify three types of entry points:

[2] The offset is typically used to get the address of a member of an aggregate type. Using an offset to get the address of another entity is unsafe because the compiler and linker can reorder entities, so normally we can assume no code intends such an effect.

[3] The exceptions are string sections and handwritten assembly.

Program entry point. A program entry point is where the execution of a process begins. For a C/C++ program, it is the startup code provided by the C library and identifiable by a platform-dependent special name. The startup code calls the main function and code which needs to run outside main such as constructors of global C++ objects.

Dynamic binding of shared libraries. An executable or shared library, or dynamic shared object (DSO), may use entities in another DSO by linking against that DSO, in which case the relations are reflected in the edge set E. It may also load a DSO at runtime and use an entity in it. We need to be able to predict statically which entities may be used in such manner.

In many cases, the caller uses a string literal as the name for the entity to be dynamically bound to. Therefore, we scan all read-only data sections in all relocatable object files, and if a symbol name s is found, the name is assumed to be used for dynamic binding at some point and all sections in $Sec(s)$ are marked as entry points.

The above analysis does not cover strings that are composed at run time. In Android, such strings are mainly related to the JNI, which enables Java code to call native code written in C/C++, and vice versa[4]. When Java calls a native function, the Java virtual machine (JVM) dynamically binds to the function in a DSO. The JVM recognizes some special name patterns[4] in the DSO, so we mark sections containing functions of those patterns as entry points. Functions in the DSO can also be "registered" to the JVM, but their addresses are already taken by a function recognized by the JVM which calls the register function.

The use of dynamic binding is relatively uncommon in Android. We searched all source code for use of relevant functions, and determined there is only one additional use of dynamic binding[5] not covered by the above analyses. After adding several related sections to set R of entry points, we conclude that set R now contains all sections that may be dynamically bound to.

The method used here is very conservative, in order to guarantee zero false negative at the cost at possibly a large number of false positives. We will demonstrate in Section V that the results are still very useful despite the false positives.

Special sections. Sections with special names (not beginning with `.text`, `.data`, `.rodata` or `.bss`) have special meanings and can be used in a way not reflected by any edge, so they are also added to set R,

TABLE II
PARAMETERS OF EXTENDED CALL GRAPH G.

n	286,893	R^*	17,618
m	25,637,355	L^*	92,978
D_i	17,042	I^*	36,944
D_o	8,709	S	37,003
D_a	89	S_m	249,728

C. Analysis of ECG

In order to achieve a global understanding of the ECG of Android system, first we present some basic parameters of the graph $G = (V, E, R)$.

Table II presents some basic parameters that help achieve a global understanding of the ECG, there are $n = 286893$ vertices and $m = 25637355$ edges in this ECG. Among those, the highest indegree[6] is $D_i = 17042$ while the highest outdegree[7] is $D_o = 8709$, and the average degree is $D_a = 89$. We define "root"(R^*) as vertices with only outdegree, "leaf"(L^*) as those with only indegree and "isolated"(I^*) as those with neither, then we get as many as 17618 "root", 92978 "leaf" and 36944 "isolated" vertices, meaning that there are plenty of vertices that do not have much interaction with the rest of the graph, which provides the potential for the optimization discussed in later sections.

Furthermore, the graph can be divided into $S = 37003$ connected components, so the complete graph G is given by $G = \cup_i G_i$. The largest subgraph actually contains $S_m = 249728$ vertices, more than 87.0% of the whole graph, leaving the rest 37002 subgraphs extremely small and isolated, each with no more than 10 vertices.

Typical complex networks often exhibit the two basic features, the scale-free (SF) topology[18] and small-world (SW) structure[19]. The SF topology states that the proportion of vertex $P(k)$ which have k degrees decays as a power law $P(k) \approx C * k^{-\gamma}$. The SW structure refers to the fact that network tends to have small average path length between vertices along with a large clustering.

The degree distribution of the ECG is shown in Fig 3, represented in the cumulative way:

$$P_{>k} = \sum_{k'>k} P(k').$$

Fig 3 shows that ECG displays scaling, with estimated exponents $\gamma_{in} \approx 0.48 \pm 0.05$ for indegree and $\gamma_{out} \approx 0.65 \pm 0.05$ for outdegree.

[4] `JNI_OnLoad`, `JNI_OnUnload` and `Java_ClassName_FunctionName`

[5] Related to ALSA sound library.

[6] The section with the highest indegree contains `WTF::fastFree`, a wrapper of `free` in WebKit.

[7] The section with the highest outdegree contains the process startup code.

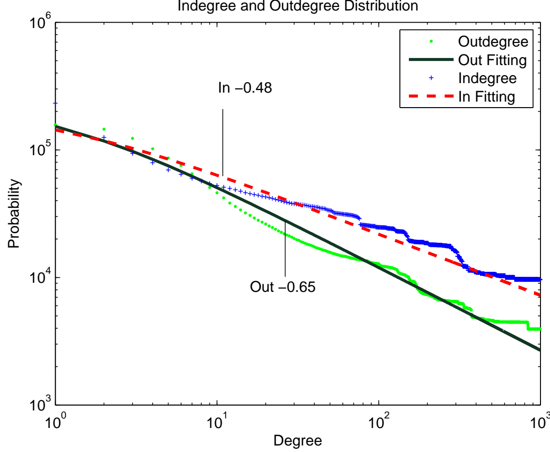

Fig. 3. Log-log plot of cumulative indegree and outdegree distribution for vertices in call graph G.

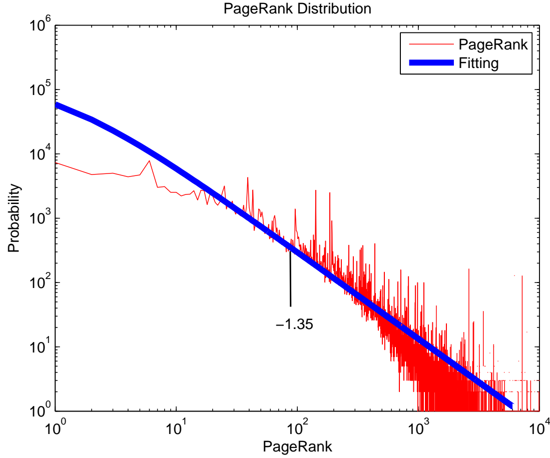

Fig. 4. The importance distribution for vertices in the ECG using PageRank algorithm. Notice its SF property.

We also adopt the PageRank Algorithm[20], [21] to evaluate the importance of the vertices in extended call graph G, and find that the result displays similar scaling property. PageRank Algorithm is originally used to measure the relative importance of the vertices within a set by assigning a numerical weighting to each element. Generally speaking, important vertices tend to have higher degrees and are likely to be connected to vertices which are also important. Figure 4 suggests that the importance of vertices exhibits a scaling feature with exponent $\gamma \approx 1.35 \pm 0.05$. The entities with top 10 PageRank are displayed in Table III. It suggests that there exists a high correlation between the PageRank and indegree/outdegree of a vertex.

The entity names given in the table clearly support this, because the top 6 are all related to dynamic memory allocation, which are fundamental and frequently used. The top function `WTF::fastFree` is a wrapper of `free`. The `free` functions in dynamic linker and libc have equal indegree and PageRank because $Use'(S)$ is used in place of $Use(S)$ to compute E (See Sec III-B). At most call sites, the one in dynamic linker is in $Use'(S) - Use(S)$, but our implementation is not yet able to reliably determine this. It is also noticeable that memory deallocation (`free`) has higher PageRank than allocation (e.g. `malloc`). The reason is that the importance of memory allocation is divided among `malloc`, `realloc` and `calloc`. However, data objects `__libc_malloc_dispatch` and `__libc_malloc_dispatch_default` gain high PageRanks despite low degrees, because they are used by all three allocation functions. Additionally, 7 of the top 10 are from bionic, which is the fundamental library

Another basic feature common to complex network is small-world structure. Normally, the small-world effect is defined by the mean geodesic (i.e., shortest) distance l between vertex pairs:

$$l = \frac{1}{\frac{1}{2}n(n-1)} \sum_{i>j} d_{ij},$$

where d_{ij} is the geodesic distance from vertex v_i to v_j. However, our ECG contains many isolated vertices, which have no path to any other vertex. In such case, the value of l would be infinite. To avoid this problem we use the following formula:

$$l^{*-1} = \frac{1}{\frac{1}{2}n(n-1)} \sum_{i>j} d_{ij}^{-1},$$

so that infinite values of d_{ij} contribute a zero to the numerator of the fraction.

We use a breadth-first search algorithm to calculate the distances from a given vertex S to every other vertices. It starts from vertex S, which is at level 0. In the first stage, we visit all the vertices that are at the distance of one edge away. When we visit there, we paint as "visited," the vertices adjacent to the start vertex S, these vertices are placed into level 1. In the second stage, we visit all the new vertices we can reach at the distance of two edges away from the source vertex S. These new vertices, which are adjacent to level 1 vertices and not previously assigned to a level, are placed into level 2, and so on. The BFS terminates when every vertex has been visited. After we have calculate the distances between any pair of vertices, we can conclude that the mean geodesic distance l^* of G is 5.94.

We can conclude from the parameters shown above that our extended call graph G is a typical complex network which exhibits scale-free feature and small-world paradigm. And despite that there are some highly

TABLE III
PAGERANK(E-3) RESULT TOP 10.

Num	PR	Out	In	Entity	Package/Module
1	7.63	4	17,042	*WTF::fastFree*	*webkit/libjs*
2	5.08	7	4,810	*free*	*bionic/linker*
3	5.08	3	4,810	*free*	*bionic/libc*
4	3.94	4	9,818	*operator delete*	*bionic/libstdc++*
5	3.94	2	12	*__libc_malloc_dispatch*	*bionic/libc*
6	3.32	10	3	*__libc_malloc_default_dispatch*	*bionic/libc*
7	3.03	27	5,532	*WTF::ThreadCondition::broadcast*	*webkit/libjs*
8	2.89	4	576	*wcscpy*	*bionic/libc_common*
9	2.66	4	2,748	*Effect_configure*	*frameworks/libbundlewrapper*
10	2.33	0	142	*__get_tls*	*bionic/libc_common*

isolated vertices, the whole graph G in fact has an interconnected kernel which contains 87.0% of all the vertices. At the same time, the PageRank result also displays scale-free property, which means that there exist a lot of unimportant vertices. From this perspective, We can conjecture that in Android operating system, which is a real-world system, there are a number of entities which are actually never called or used by others. More importantly, these entities take up much system space and may affect system performance. So we have a very good reason to assume that by eliminating them we can achieve a smaller system code size and an enhanced performance.

IV. DEAD CODE ELIMINATION

A. *Significant Subgraph of ECG*

The analysis of the ECG in the previous section has implied that there exist a number of sections that can be safely eliminated from the system. In order to identify them more accurately, we first define the significant subgraph of the ECG.

Definition 8. The *significant subgraph* of ECG $G = (V, E, R)$ is $G_e = (V_e, E_e, R)$, where V_e is the set of *significant vertices* or *significant sections*, $V_e = \bigcup_{S \in R} Desc(S)$. $Desc(S)$ is the set of all descendants of S in the ECG, including S itself, and E_e is the set of *significant edges*, $E_e = E \cap (V_e \times V_e)$.

According to the definition, a significant section is either an entry point or reachable from an entry point. Therefore, V_e is the set of all entities that must be retained in the system, and the rest, i.e. those in $V - V_e$ can be eliminated. The algorithm to find V_e is a straightforward graph search.

B. *Rewriting Relocatable Object Files*

After finding the set of sections to be eliminated, $V - V_e$, we can rewrite the relocatable object files so that those sections will not be linked into the final binaries. The GNU linker, which is the default linker for Android, has a feature known as section "garbage collection." If enabled, this feature causes the linker to discard unreachable sections at link time[22]. This functionality, combined with the "section-per-entity" compiler feature (see Section III-B), is capable of eliminating unused entities at the binary level without the complex interprocedural analysis[22]. In our method, the ECG is used to allow similar dead code elimination at the system level.

After finding the ECG and its significant subgraph, we use binary rewriting to modify the relocatable object files. Specifically, each section $S \in V - V_e$ is marked in a way that enables the linker to know it is safe to be eliminated.

Step 1. For each externally visible entity in a section in $V - V_e$, change the symbol table to make it "hidden."

Step 2. For each section in $V - V_e$, clear its relocation data so that it no longer uses any other section.

Step 3. For each external reference in the symbol table, remove it if it is no longer useful due to Step 2. This operation changes indices of other symbol table entries, so it is necessary to revisit all sections and update references to the symbol table.

Step 4. Re-link all shared libraries and executables.

It should be noted that Steps 2 and 3 are not redundant. The GNU linker performs the section garbage collection after merging symbol tables. As a result, external references from eliminated sections are retained in the symbol table of the final binary if Steps 2 and 3 are skipped. It is not a problem in normal use of the section garbage collection feature except that the binary is slightly larger than ideal. In the system-wide optimization, however, it causes actual problems. We have an example from Android: Function `_WLocale_strcmp` in libstlport calls function `wcsncpy` in libc, but both are unreachable and eliminated. If Steps 2 and 3 are skipped, the dynamic linking symbol table of libstlport still contains an external reference to `wcsncpy`, though both functions

TABLE IV
PARAMETERS OF EXTENDED CALL GRAPH G AND G_e.

Para	G	G_e	Diff	Percent
n	286,893	218,885	-68,008	-23.7%
m	2.56E+7	2.49E+7	-697,336	-2.7%
D_i	17,042	14,483	-2,559	-15.0%
D_o	8709	8709	0	0.0%
D_a	89	113	+24	+27.0%
R^*	17,618	1,198	-16,420	-93.2%
L^*	92,978	87,733	-5,245	-5.6%
I^*	36,944	167	-36,777	-99.5%
S	37,003	176	-36,827	-99.5%
S_m	249,728	218,691	-31,037	-12.4%
l^*	5.94	4.50	-1.44	-24.2%

TABLE V
BASIC PARAMETERS OF PACKAGE *bionic* IN G AND G_e.

Para	G	G_e	Diff	Percent
n	900	567	-333	-37.0%
m	54,088	45,381	-8,707	-16.1%
D_i	4,811	4,134	-677	-14.1%
D_o	46	44	-2	-4.4%
D_a	60.1	80.0	+19.9	+33.1%

have been eliminated. The linker then refuses to accept libstlport as a valid input to create other binaries.

V. IMPLEMENTATION AND EVALUATION

Android mainly runs on ARM, but has also been ported to several other architectures. The Android-x86 project[23] has been releasing x86 ports since 2009. Its native code is, by default, built by a modified version of the GNU toolchain[24][25] shipped with the source code. The experiments presented in this section are performed on the honeycomb-x86 branch (version 3.2.2) of Android-x86[23], pulled from repositories on Feb 15, 2012. The build target is "eeepc-eng." The built images are tested on an ASUS Eee PC[26].

The tools to read the relocatable object files, construct the ECG, find the set of significant sections and rewrite the relocatable object files are implemented in C++ and tested on a Linux system. Elfutils[27] is used to handle details in ELF. The tools used to perform the complex networks analysis in Section III-C are implemented in C/OpenMP and run on NUMA-SMP server.

A. Call Graph Connectivity Improved

Table IV shows the basic parameters of the ECG and significant subgraph G_e.

The average links per vertex D_a is raised by 27.0%, meaning that there exist far few useless vertices. The percentage of vertices in the largest connected subgraph is increased from 87.0% to 99.9%. And most of the smaller isolated subgraphs are eliminated, leaving only 176 better connected ones. In a word, parameters improve surprisingly, making G_e much better connected and centralized.

When we examine other parameters, as many as 68008 vertices have been eliminated from graph G, leaving $n = 218885$. Among "root", "leaf" and "isolated", there is a major decline in the number of "isolated" and "root" vertices. As a result, the connectivity of graph G_e is greatly improved. D_o remains while D_i decreases a little bit not because the vertex with D_i is eliminated but because some other vertices linking to it were eliminated.

After the optimization, the graph G_e still displays scale-free and small-world features, the exponents γ_{In} and γ_{Out} remain almost the same as G.

l^* after optimization decreases by 24.2%. It is because according to the calculation formula,

$$l^{*-1} = \frac{1}{\frac{1}{2}n(n-1)} \sum_{i>j} d_{ij}^{-1},$$

before optimization those unconnected vertex pairs contribute a zero to the numerator of the fraction but they are still counted in the denominator $\frac{1}{2}n(n+1)$, while after optimization they are eliminated and make no contribution to the formula at all. In this way, l^* decreases, which means the connectivity and transitivity of the graph is increased.

Table V shows the basic parameters of extended call graph inside *bionic*. Being one of the most important packages of Android system, *bionic* changes drastically after the optimization. n falls significantly by 37% while D_a raises noticeably by 33% due to the elimination of unreachable vertices which have links to *bionic*.

B. Code Size Reduced

Table VI displays the code size of the default and different optimized builds of Android-x86. Included in the data are executables and shared libraries, excluding the Linux kernel. The "dyn*" column includes dynamic linker symbol tables (dynsym) and string tables for dynamic linking (dynstr). The "other" column includes regular symbol tables (symtab) and string tables (strtab), and other sections which are normally very small. "Total" is larger than "file size" because BSS sections, which are included in "total", occupy memory at run time but no space in disk storage. The text and file size columns are also displayed in Figure 5.

In the default build (DB), Android-x86 is built without any modification. Android-x86 by default strips debugging information from binaries.

In the FS build, we strip all information that are unneeded at run time. Specifically, regular symbol tables and string tables are stripped, but those needed for

TABLE VI
COMPARISON OF CODE SIZE OF ALL ELF FILES (EXCLUDING THE LINUX KERNEL) IN ANDROID-X86 (IN KB)

Configuration	Sections							File size
	text	rodata	data	bss	dyn*	others	total	
default build (DB)	48,579 (100%)	7,682 (100%)	2,668 (100%)	4,369 (100%)	3,865 (100%)	16,116 (100%)	83,280 (100%)	79,335 (100%)
fully stripped (FS)	48,579 (-0.0%)	7,682 (-0.0%)	2,668 (-0.0%)	4,369 (-0.0%)	3,865 (-0.0%)	3,218 (-80.0%)	70,381 (-15.5%)	66,402 (-16.3%)
binary-wide optimization (BWO)	44,852 (-7.7%)	7,604 (-1.0%)	2,622 (-1.7%)	4,251 (-2.7%)	3,459 (-10.5%)	3,133 (-80.6%)	65,921 (-20.8%)	62,067 (-21.8%)
system-wide optimization (SWO)	42,099 (-13.3%)	7,316 (-4.8%)	2,555 (-4.2%)	4,239 (-3.0%)	2,798 (-27.6%)	2,949 (-81.7%)	61,957 (-25.6%)	58,121 (-26.7%)

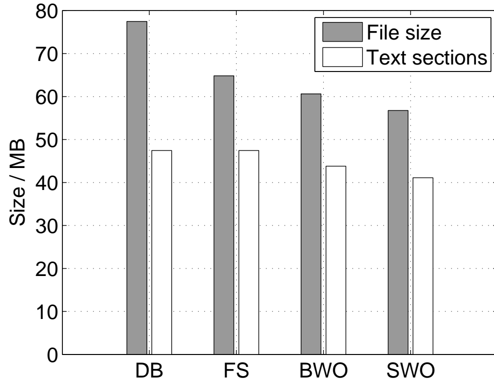

Fig. 5. Comparison of size of files and text sections.

dynamic linking must be preserved; comment sections, which are normally small and contain information such as compiler version, are stripped as well.

In the BWO build, the “section-per-entity” feature in GCC and section garbage collection in the GNU linker are enabled. In addition, all sections unneeded at run time are stripped (comment sections, regular symbol tables and string tables, excluding those for dynamic linking). The BWO build is the smallest we are able to achieve with the existing functionalities of the toolchain. Compared with the default build, the file size is reduced by 21.8%, and the size of text sections by 7.7%.

In the SWO build, the system-wide dead code elimination described in Sec IV is enabled, and a further reduction of 6.4% in file size and 6.1% in text sections is achieved. Compared with the default build, the file size is reduced by 26.7%, and the text sections 13.3%.

Table VII displays the size data for bionic. As we can see from the results, bionic still contains a considerable percentage of dead code though it is specifically written and optimized for Android.

C. Correctness Preserved

We use Android Monkey[28] to evaluate the correctness of the optimized system. As a default utility of Android, Monkey generates pseudo-random streams of user events such as clicks, touches and gestures, as well as a number of system-level events[28]. Originally intended for stress test, it has now gained a wider range of use, e.g. suspicious software detection[29] and automated test case generation[30].

The optimized system is expected to respond to each event in the same way as the original system. In order to get repeatable results, we specify a fixed seed for the pseudo-random generator, and force a one-second delay between consecutive events to ensure that the process of each event is complete before the next one is sent. In our experiment, 10000 events are generated, and the original and optimized systems behave in exactly the same way and generate logs that differ only in timestamps. The test covers most of the pre-installed packages in Android.

D. Performance Improved

The method presented in this paper does no modification to the code except removing code that will never be used, so it can be expected that performance remains unchanged or is slightly improved due to reduced I/O, memory footprint and cache miss. Experiments have confirmed this.

A number of performance benchmarks[31] for Android have emerged in recent years, such as AnTuTu-Benchmark[32], Smartbench[33] and AndroBench[34]. We choose AnTuTu-Benchmark because it tests many aspects of the system, including memory performance, CPU integer performance, CPU floating-point performance, 2D/3D graphics performance, database I/O and SD card I/O. Experiments with AnTuTu-Benchmark 2.8.3 on an ASUS Eee PC show an improvement by 1.3% in floating-point performance, and no statistically significant changes in other figures. No performance degradation is observed.

Additionally, AnTuTu-Benchmark is a complex application which makes use of JNI. Its ability to run on the optimized system is also an evidence of the correctness of our optimization.

TABLE VII
COMPARISON OF CODE SIZE OF BIONIC (IN KB)

Configuration	Sections							File size
	text	rodata	data	bss	dyn*	others	total	
default build (DB)	322 (100%)	29 (100%)	12 (100%)	45 (100%)	27 (100%)	61 (100%)	496 (100%)	452 (100%)
fully stripped (FS)	322 (-0.0%)	29 (-0.0%)	12 (-0.0%)	45 (-0.0%)	27 (-0.0%)	19 (-68.9%)	454 (-8.5%)	410 (-9.3%)
binary-wide optimization (BWO)	322 (-0.0%)	29 (-0.0%)	12 (-0.0%)	45 (-0.0%)	27 (-0.0%)	19 (-68.9%)	453 (-8.7%)	410 (-9.3%)
system-wide optimization (SWO)	283 (-12.1%)	27 (-6.9%)	11 (-8.3%)	44 (-2.2%)	20 (-25.9%)	17 (-72.1%)	402 (-19.0%)	361 (-20.1%)

VI. RELATED WORK

Complex Networks. With a sound theoretical basis and a variety of applications, complex networks offer a perfect nonlinear abstraction when analyzing applications to real-world problems[8]. Empirical study of real-world networks such as computer networks and social networks largely inspired the research of complex network. In the area of software, massive work has been dedicated to survey the relationship between complex networks and the graph constructed by software components, e.g. packages, files, class, functions, etc. Valverde et al.[10] studied the topology and hierarchical relationships among components of various software packages, and found that all these networks exhibit the same organization pattern such as scale-free topology and small-world property. David Wood et al.[35] analyzed software collaboration graphs for open-source software projects written in Java and found the graphs produced at package, class and method levels all display scale-free properties. Alessandro et al.[36] investigated network properties of four widely used large computer programs, the Linux kernel, Mozilla, XFree86 and Gimp from a source file and header file dependency level and found similar results. For the class dependency level, Lovro Šubelj et al.[37] discovered that complex networks not only follow SF and SW phenomena, but also have the property of community structure. C. R. Myers[9] studied class graphs of VTK, CVS and abiWord, the call graphs of the Linux Kernel, MySQL and XMNS, found their scale-free, small-world topologies and presented a simple model of software system evolution based on refactoring processes. Despite the various researches on network of open-source software, most of these works focus only on single software, or on the package level. None of them handles function-level relations in multilingual complex systems as Android.

System optimizations. Early optimizers attempted to generate better code for a single basic block or a single function. The optimizer had little information to rely on, and the ability to produce better code was limited. With the development of inter-procedural analysis, it became possible to optimize a compilation unit and a binary as a whole. Later, methods were developed to use even more information, such as using information from all sources in a multiple-binary system. Ho et al.[38] used cross-module optimization to eliminate indirect addressing and reduce the run-time overhead of dynamically-linked programs. Acharya and Saltz[39] used inter-procedural analysis to identify names that must be linked at different sites and use this information to optimize dynamic linking for mobile programs. A scalable cross-module optimization proposed by Ayers et al.[40] is also in this category. Binary rewriting has been used to optimize different applications including a wide range of ARM applications by De Bus et al.[41] and the Linux kernel by Chanet et al.[42] In addition to optimizations based on information available from the source code, feedback directed optimization has been developed to allow the optimizer to use external information. Rus et al.[43] used it to optimize string operations on large data centers. There are also works to optimize the Java code of Android. ProGuard[44] is a free Java class file shrinker, optimizer, obfuscator, and preverifier. It detects and removes unused classes, fields, methods, and attributes. It optimizes bytecode and removes unused instructions. In this paper, we use binary rewriting to optimize the native code of Android except those in the Linux kernel.

VII. CONCLUSION AND FUTURE WORK

The extended call graph (ECG) presented in this paper can model and analysis the multilingual complex system Android as complex networks very well. We identified that the ECG of Android system exhibits scale-free and small-world feature.

System-wide dead code elimination optimization can be performed based on these results. Most “isolated” vertices and subgraphs are found not to be in the significant subgraph and are eliminated to build a better connected graph. The percentage of vertices in the largest

connected subgraph is greatly increased. At the same time, the average degree of vertex is raised by 27.0%. It has also achieved about 26.7% code size reduction and up to 1.3% speedup in floating-point computation. Even the dedicated bionic libc still has 20.1% dead code which are eliminated by our method.

This work not only provides a foundation of optimizing Android operating system for both performance and code size, but also helps us to understand and develop the complex software system more efficiently. Although Android-x86 3.2.2 is taken as the example to demonstrate our method in this paper, we believe that other versions of Android and other mobile operating systems developed in C/C++ and assembly languages can also be modeled, analyzed and optimized following the same approach.

In the future we will continue to improve the construction of extended call graph. We will use package dependency relations and other information to further reduce the number of false edges and entry points.

In addition, we plan to explore additional code and data layout optimizations for Android that are both practical for deployment and strong enough for performance improvement.

In the aspect of complex network, there are still various features to explore. Further studies on how these complex network features actually affect the performance of software systems may enlighten us to develop new optimization technology.

VIII. Acknowledgement

We would like to thank Mr. Sun Chan of Intel, Dr. Gang Tan of Lehigh University and Prof. Pen-Chung Yew of the University of Minnesota for useful discussions about software analysis in this paper. We also thank Dr. Zhiyuan Liu of Tsinghua University for the helpful discussions about complex network. Prof. Wei Xue of Tsinghua University provided a NUMA-SMP computer to support complex network computations. Pengqi Cheng helped us to build the ECG network. Junjie Mao helped us to find all the JNI interface out. The authors are supported in part by National Natural Science Foundation of China (No. 61170051, No. 61272086), Hi-Tech Research and Development Program of China (No. 2009AA011902), National Core-High-Base Project of China (No. 2012ZX01039-004-003), and Research Foundation of Tsinghua University (No. 2010Z05097). Any opinions, findings, and contributions contained in this document are those of the authors and do not reflect the views of these agencies.

References

[1] Google, “Android project webpage,” http://www.android.com, 2012.

[2] R. Cozza, C. Milanesi, A. Zimmermann, T. H. Nguyen, H. J. De La Vergne, S. Shen, A. Gupta, A. Sato, C. Lu, and D. Glenn, “Market share: Mobile devices by region and country, 4q11 and 2011,” *Gartner*, Feb 2012.

[3] “What is android? — android developers,” http://developer.android.com/guide/basics/what-is-android.html, 2012.

[4] S. Liang, *Java Native Interface: Programmer’s Guide and Reference*, 1st ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[5] D. A. Wheeler, “Sloccount project webpage,” http://www.dwheeler.com/sloccount, 2012.

[6] F. Maker and Y.-H. Chan, “A survey on Android vs. Linux,” *University of California*, 2009.

[7] “Glibc, the GNU C library webpage,” http://www.gnu.org/software/libc/, 2012.

[8] L. da Fontoura Costa, O. N. Oliveira Jr., G. Travieso, F. A. Rodrigues, P. R. V. Boas, L. Antiqueira, M. P. Viana, and L. E. C. Rocha, “Analyzing and modeling real-world phenomena with complex networks: a survey of applications,” *Advances in Physics*, vol. 60, no. 3, pp. 329–412, 2011.

[9] C. R. Myers, “Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs,” *Physical review E*, vol. 68, no. 4, pp. 046 116–046 130, 2003.

[10] S. Valverde and R. V. Solé, “Hierarchical small worlds in software architecture,” in *Dynamics of Continuous Discrete and Impulsive Systems*, ser. Series B, Applications and Algorithms, 2007, pp. 1–11.

[11] N. LaBelle and E. Wallingford, “Inter-package dependency networks in open-source software,” in *arXiv*, ser. cs.SE/0411096, 2004.

[12] T. Maillart, D. Sornette, S. Spaeth, and G. V. Krogh, “Empirical tests of zipf’s law mechanism in open source linux distribution,” *Physical review letters*, vol. 101, no. 21, 2008.

[13] S. K. Debray, W. Evans, R. Muth, and B. De Sutter, “Compiler techniques for code compaction,” *ACM Trans. Program. Lang. Syst.*, vol. 22, no. 2, pp. 378–415, Mar. 2000. [Online]. Available: http://doi.acm.org/10.1145/349214.349233

[14] “GCC link time optimization webpage,” http://gcc.gnu.org/wiki/LinkTimeOptimization, 2012.

[15] G. Chakrabarti and F. Chow, “Structure layout optimizations in the Open64 compiler: design, implementation and measurements,” in *Open64 Workshop at the International Symposium on Code Generation and Optimization*, ser. CGO ’08, Boston, MA, USA, 2008.

[16] C. Lattner, “Introduction to the LLVM compiler infrastructure,” in *Itanium Conference and Expo*, San Jose, CA, USA, 2006.

[17] TIS Committee, *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2*, May 1995.

[18] R. Albert and A.-L. Barabási, “Statistical mechanics of complex networks,” *Reviews of Modern Physics*, vol. 74, 2002.

[19] M. E. J. Newman, “Models of the small world,” *J. stat. phys.*, vol. 101, no. 3/4, pp. 819–841, 2000.

[20] L. Page, S. Brin, R. Motwani, and T. Winograd, “The pagerank citation ranking: Bringing order to the web.” Stanford InfoLab, Technical Report 1999-66, November 1999, previous number = SIDL-WP-1999-0120. [Online]. Available: http://ilpubs.stanford.edu:8090/422/

[21] A. Alon and M. Tennenholtz, “Ranking systems: The pagerank axioms,” in *Proceedings of the 6th ACM conference on Electronic commerce*, 2005.

[22] D. Chanet, B. De Sutter, B. De Bus, L. Van Put, and K. De Bosschere, “System-wide compaction and specialization of the Linux kernel,” in *Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, ser. LCTES ’05. New York,

NY, USA: ACM, 2005, pp. 95–104. [Online]. Available: http://doi.acm.org/10.1145/1065910.1065925

[23] "Android-x86 project," http://www.android-x86.org, 2012.

[24] "GNU binutils webpage," http://www.gnu.org/software/binutils/, 2012.

[25] R. Stallman, "Using and porting the GNU compiler collection," *M.I.T. Artificial Intelligence Laboratory*, 2001.

[26] ASUSTek, "Asus eee pc 1000ha," http://eee.asus.com, 2012.

[27] "elfutils home page," https://fedorahosted.org/elfutils/, 2012.

[28] "UI/application exerciser monkey," http://developer.android.com/guide/developing/tools/monkey.html, 2012.

[29] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: behavior-based malware detection system for android," in *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, ser. SPSM '11. New York, NY, USA: ACM, 2011, pp. 15–26. [Online]. Available: http://doi.acm.org/10.1145/2046614.2046619

[30] C. Hu and I. Neamtiu, "Automating gui testing for android applications," in *Proceedings of the 6th International Workshop on Automation of Software Test*, ser. AST '11. New York, NY, USA: ACM, 2011, pp. 77–83. [Online]. Available: http://doi.acm.org/10.1145/1982595.1982612

[31] H.-J. Yoon, "A study on the performance of Android platform," *International Journal on Computer Science and Engineering*, vol. 4, no. 4, pp. 532–537, 2012.

[32] AnTuTu Labs, "Antutu benchmark project webpage," http://www.antutulabs.com/AnTuTu-Benchmark, 2012.

[33] "Smartphone benchmarks webpage," http://www.smartphonebenchmarks.com, 2012.

[34] J.-M. Kim and J.-S. Kim, "Androbench: Benchmarking the storage performance of android-based mobile devices," *Advances in Intelligent and Soft Computing*, vol. 133, pp. 667–674, 2012.

[35] D. Hyland-Wood, D. Carrington, and S. Kaplan, "Scale-free nature of java software package, class and method collaboration graphs," *Technical Report*, vol. No. TR-MS1286, 2006.

[36] A. P. S. de Moura, Y.-C. Lai, and A. E. Motter, "Signatures of small-world and scale-free properties in large computer programs," *Physical Review E*, vol. 68, pp. 017 102–017 105, 2003.

[37] L. Šubelj and M. Bajec, "Community structure of complex software systems: Analysis and applications," *Physica A*, vol. 390, pp. 2968–2975, 2011.

[38] W. W. Ho, W.-C. Chang, and L. H. Leung, "Optimizing the performance of dynamically-linked programs," in *Proceedings of the USENIX 1995 Technical Conference Proceedings*, ser. TCON'95. Berkeley, CA, USA: USENIX Association, 1995, pp. 19–19. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267411.1267430

[39] A. Acharya and J. H. Saltz, "Dynamic linking for mobile programs," in *Selected Presentations and Invited Papers Second International Workshop on Mobile Object Systems - Towards the Programmable Internet*, ser. MOS '96. London, UK, UK: Springer-Verlag, 1997, pp. 245–262. [Online]. Available: http://dl.acm.org/citation.cfm?id=648087.747337

[40] A. Ayers, S. de Jong, J. Peyton, and R. Schooler, "Scalable cross-module optimization," in *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, ser. PLDI '98. New York, NY, USA: ACM, 1998, pp. 301–312. [Online]. Available: http://doi.acm.org/10.1145/277650.277745

[41] B. De Bus, B. De Sutter, L. Van Put, D. Chanet, and K. De Bosschere, "Link-time optimization of arm binaries," in *Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, ser. LCTES '04. New York, NY, USA: ACM, 2004, pp. 211–220. [Online]. Available: http://doi.acm.org/10.1145/997163.997194

[42] D. Chanet, B. De Sutter, B. De Bus, L. Van Put, and K. De Bosschere, "Automated reduction of the memory footprint of the linux kernel," *ACM Trans. Embed. Comput. Syst.*, vol. 6, no. 4, Sep. 2007. [Online]. Available: http://doi.acm.org/10.1145/1274858.1274861

[43] S. Rus, R. Ashok, and D. X. Li, "Automated locality optimization based on the reuse distance of string operations," in *Proceedings of the CGO 2011, The 9th International Symposium on Code Generation and Optimization, Chamonix, France, April 2-6, 2011.* IEEE, 2011, pp. 181–190.

[44] E. Lafortune, "Proguard project webpage," http://proguard.sourceforge.net/, 2012.