On-Device Objective-C Application Optimization Framework for High-Performance Mobile Processors

Garo Bournoutian
University of California, San Diego
9500 Gilman Dr. #0404 La Jolla, CA 92093–0404
Email: garo@cs.ucsd.edu

Alex Orailoglu
University of California, San Diego
9500 Gilman Dr. #0404 La Jolla, CA 92093–0404
Email: alex@cs.ucsd.edu

***Abstract*—Smartphones provide applications that are increasingly similar to those of interactive desktop programs, providing rich graphics and animations. To simplify the creation of these interactive applications, mobile operating systems employ high-level object-oriented programming languages and shared libraries to manipulate the device's peripherals and provide common user-interface frameworks. The presence of dynamic dispatch and polymorphism allows for robust and extensible application coding. Unfortunately, the presence of dynamic dispatch also introduces significant overheads during method calls, which directly impact execution time. Furthermore, since these applications rely heavily on shared libraries and helper routines, the quantity of these method calls is higher than those found in typical desktop-based programs. Optimizing these method calls centrally before consumers download the application onto a given phone is exacerbated due to the large diversity of hardware and operating system versions that the application could run on. This paper proposes a methodology to tailor a given Objective-C application and its associated device-specific shared library codebase using on-device post-compilation code optimization and transformation. In doing so, many polymorphic sites can be resolved statically, improving the overall application performance.**

I. Introduction

The prevalence of mobile processors has grown significantly over the last few years. At the current rate, mobile processors are becoming increasingly ubiquitous throughout our society, resulting in a diverse range of applications that will be expected to run on these devices. State-of-the-art smartphones have evolved to the level of having feature-rich applications comparable to those of interactive desktop programs, providing high-quality visual and auditory experiences. These mobile processors are becoming increasingly complex in order to respond to this more diverse and demanding application base.

From the application perspective, complexity is also growing within the mobile domain. A decade ago, mobile phone applications consisted of a few pre-bundled, hand-optimized programs specifically designed for a given device. Today, there is a vast quantity of applications available within mobile marketplaces; Apple's *App Store*, for example, had over 775,000 applications available for download at the start of 2013 [1]. Furthermore, mobile applications are now written in high-level, object-oriented programming languages such as Objective-C or Java. These applications tend to be highly interactive, providing visual, auditory, and haptic feedback, as well as leveraging inputs from numerous sensors such as touch, location, near-field communication, and even eye tracking. To simplify the creation of these interactive applications, mobile operating systems provide foundation libraries to manipulate the device's peripherals and provide common user-interface frameworks. Given this, much of the instruction code being executed will be coming from these shared libraries and helper routines, in addition to the application's own code.

One powerful feature enabled by using a high-level, object-oriented language is dynamic dispatch. Dynamic dispatch allows for robust and extensible application coding, allowing multiple classes to contain different implementations of the same method, with actual method selection occurring dynamically based on the particular run-time instance it is called upon. Unfortunately, the presence of dynamic dispatch can introduce significant overheads during method calls, which can directly impact execution time [2]. It has also been observed that smartphone applications suffer from increased code size and sparseness [3], and encounter higher branch mispredictions and instruction cache misses due to lack of instruction locality. As this paper will demonstrate, dynamic dispatch exacts a heavy toll due to the numerous control flow changes that occur from the large quantity of method call polymorphic sites.

To make matters worse, there is a large diversity of hardware and operating systems a given application may run on. Often a single instance of the application is compiled and uploaded to the marketplace, and then is able to be downloaded and run on many different devices and operating system versions. When the application is loaded by the operating system, dynamic linking and variable offset tables are used to hook into the device-specific library code. Furthermore, some methods the application may call will be stubbed out for a particular model of hardware. For example, if the application calls a method to query the compass, but the specific device being utilized does not physically have a compass, the call will simply return without executing the actual sensor processing instructions. As one can see, attempting to optimize these polymorphic method calls centrally before consumers download the application onto a given phone is hindered by the variability and diversity of the final execution platform.

In this paper, we propose a novel approach to deal with the unique characteristics of mobile processors and provide a methodology to tailor a given Objective-C application and its associated device-specific shared library codebase using on-device post-compilation code optimization and transformation. While an application is still compiled normally into binary code at the time it is added to the application marketplace, we propose also garnering and embedding class hierarchy metadata in the static deliverable. Once the application is downloaded onto a specific device, a novel second pass of code optimization is then performed combining the application's embedded metadata with the local device's framework metadata to identify polymorphic sites that can be replaced with static method

978–3–9815370–2–4/DATE14/©2014 EDAA

calls. This innovative approach widens the information channel of the application binary and enables otherwise impossible holistic on-device optimizations. We are able to replace an average of 76.8% of dynamic dispatch sites with purely static method calls, greatly reducing the amount of code execution necessary to resolve method control flow, delivering improved overall application performance. The result will be an optimized application that encounters fewer instruction cache misses and branch mispredictions, which in turn helps alleviate overall processor power consumption leading to longer battery life.

II. Related Work

Most prior research related to object-oriented programming languages and, in particular, dynamic dispatch has focused on standard general-purpose computers and enterprise programs.

Profile-guided approaches are commonly used to enable dynamic dispatch code optimization. The *Call Chain Profile* (k-CCP) model was presented in [4]. Using this model, the predictability of receiver class distributions was presented, demonstrating that such distributions are strongly peaked and relatively stable across program inputs. They also proposed inlining run-time tests for the dominant class in order to help optimize dynamic dispatch for the strongly peaked calls. The *Festival* approach attempts to estimate the frequency of virtual method calls by applying neural network machine learning algorithms across a set of known programs and workloads to glean relationships between static code structures and actual run-time behaviors [2]. By identifying the highly recurring calls, one can enable targeted method lookup caching to help ameliorate the most problematic locations. The drawback to these profile-guided approaches is the necessity of having accurate profiling workloads. Given that mobile applications are almost always user interface driven and highly interactive, the ability to have representative workloads of dynamic execution is non-trivial.

Class Hierarchy Analysis (CHA) was proposed in [5], where for each method the set of classes for which that method is the appropriate target is determined (*applies-to* set). When a polymorphic method call is encountered, the *applies-to* sets are checked to see if there is any overlap. If not, then the polymorphic site can be replaced by a static call. This approach requires knowledge of all class hierarchies, as well as methods defined within each class. Similarly, the *SmallEiffel* compiler leverages type inference to remove polymorphic call sites and instead replace them with static bindings when possible [6]. Essentially, the entire program is analyzed and a set of all possible concrete types is identified. All living methods are duplicated and customized based on the concrete type of the target. Thus, if only a single matching method exists for a given call, it is replaced by a regular static call to the target method. Unfortunately, these approaches require full application code analysis. Since mobile applications are centrally compiled without complete knowledge of all device-specific operating system library code, this approach cannot guarantee correctness. Furthermore, once the application is compiled into binary code, extraction of class hierarchy information becomes exceedingly difficult.

On-the-fly generation of remote device-specific code using centralized system notes is presented in [7]. The *Fiji VM* does aggressive de-virtualization and inlining to mitigate method call overhead [8]. These approaches typically rely on running within a Virtual Machine (VM) to enable such JIT optimizations.

```
01  @implementation Animal : NSObject
02  -(void) makeSound {
03      /* Display string... */
04  }
05  @end
06  @implementation Cow : Animal
07  -(id) init {
08      self = [super init];                        /* 1 */
09      self->sound = @"Moo";
10      return self;
11  }
12  @end
13  @implementation Pig : Animal
14  -(id) init {
15      self = [super init];                        /* 1 */
16      self->sound = @"Oink";
17      return self;
18  }
19  @end
20  int main(int argc, const char * argv[]) {
21      NSAutoreleasePool *pool =
            [[NSAutoreleasePool alloc] init];     /* 2 */
22      Cow *myCow = [[Cow alloc] init];            /* 2 */
23      [myCow makeSound];                          /* 1 */
24      Pig *myPig = [[Pig alloc] init];            /* 2 */
25      [myPig makeSound];                          /* 1 */
26      [pool release];                             /* 1 */
27      return 0;
28  }
```

Fig. 1. Simple Objective-C Application

The domain of native-execution mobile processors introduces unique challenges related to mitigating dynamic dispatch overhead. Since mobile applications are highly interactive and frequently utilize polymorphism both within the application's local source code and externally into common foundation library code, a solution that can statically analyze the combination of the original application code and its interaction with common device-specific libraries becomes necessary.

III. Motivation

In order to illustrate the overhead of dynamic dispatch, a simple Objective-C application is shown in Figure 1. This application defines three classes, a parent class (*Animal*) that extends from the foundation base class *NSObject*, as well as two child classes (*Cow* and *Pig*) that both extend from *Animal* and override the default *init* instance method defined in *NSObject*. Objective-C uses the syntax of `[obj method:argument]` in order to send a message (identified by the *selector* `method`) to the *receiver* `obj`. Due to dynamic dispatch, the resolution of the method *selector* to the underlying C method pointer *implementation* occurs at runtime. For the sake of simplicity, assume that any methods whose implementation is not defined herein will send no subsequent messages. With this assumption, this simple application will result in the sending of 11 messages when executed, as shown in the code comments.

Unlike regular C function calls, where control flow unconditionally moves to the target function label, Objective-C message sending is quite complicated. Indeed, during runtime resolution of a message, the current class and all superclass metadata is searched to see if the *selector* exists within that class's method list. This process is very cumbersome and involves complex memory traversals. The source code for the iOS Objective-C method lookup is provided in [9].

In order to avoid repeatedly traversing frequently selected methods, a software-based method cache exists for each class to quickly map *implementations* for a given *selector*. The vast majority of lookups are able to be resolved by the software cache. Yet, even if a message lookup hits the method cache, the overhead is still substantially more than a pure C function call. Examining the cache lookup portion of the iOS *_objc_msgSend* routine [10] will show that even if the cache lookup results in an immediate cache hit (on the first index of the cache), 21 instructions are executed, 3 of which are conditional branches. Referring back to the example application in Figure 1, each of

TABLE I. RELATIVE TIME PER METHOD CALL

Type of Method Call	Intel Core i5	iPhone 4S
IMP-cached message send	1.00X	21.55X
C++ virtual function call	1.99X	3.96X
Objective-C message send	12.57X	69.46X

the 11 message calls will necessitate executing *at minimum* 21 instructions plus conditional branching. The key observation is that this behavior can have tangible consequences on the overall processor performance. Primarily, the number of additional instructions executed during each method call contributes to increased execution time.

In order to get a current perspective on the implementation overheads of both C++ virtual function calls and Objective-C message sends, a basic benchmark of these calls within the Apple Cocoa framework was conducted. Each type of method call was executed 10 billion times, and the overall run-time was captured. The normalized execution time overhead for each type of method call is shown in Table I. The measurements were taken on a regular desktop computer using an Intel Core i5 as well as on an iPhone 4S. It is interesting to note that on a desktop computer, the time overhead of C++ virtual function calls is comparable to caching the implementation pointer of an Objective-C method and using that to repeatedly send messages (IMP-cached message send). Yet, when run on a mobile processor, the overhead of even IMP-cached message sends is quite apparent. Given the architectural differences between Intel x86 and iPhone ARM-based SoC, it appears the overhead of dynamic dispatch is more salient in the latter.

In addition to dynamic dispatch increasing dynamic instruction counts, there are also repercussions in terms of the instruction cache and branch predictor performance. With regard to the instruction cache, spatial locality is paramount for ideal performance. As the control flow of the application changes, spatial locality is degraded. Every polymorphic call necessitates a jump to the dispatch instructions, wherein a cache lookup may occur entailing more control flow changes. This behavior can manifest in terms of increased L1 instruction cache misses.

Regarding branch prediction, the increase in conditional branches to deal with method cache lookups can impact overall performance. Prediction structures, such as the global history register, may become polluted with these dispatch-related conditional branches. Furthermore, most dynamic dispatch approaches rely on indirect (register-based) branching, wherein target address prediction becomes an issue. According to [11], despite specialized hardware to predict indirect jump targets, 41% of mispredictions come from indirect jumps. To exacerbate matters, those results are based on a desktop processor with specialized indirect jump prediction hardware. Most mobile processors, being area and power constrained, often employ simplified branch prediction hardware.

It becomes apparent that dynamic dispatch not only increases execution time due to executing more instructions, but also increases entropy in many of the processor's hardware predictive structures like caches and branch predictors. If possible, one would like to replace as many polymorphic sites with direct function calls in order to ameliorate these overheads.

IV. METHODOLOGY

One of the primary challenges to optimizing dynamically dispatched applications on mobile processors is the vast diversity of hardware, OS versions, and consumer applications. Mobile applications are developed in a loosely general-purpose

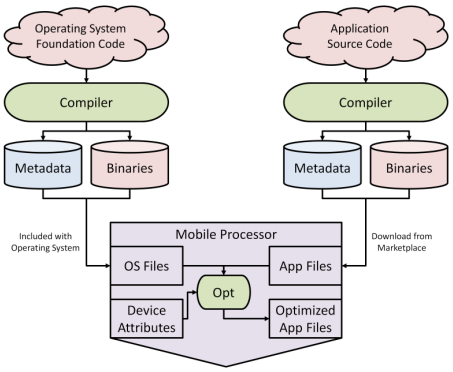

Fig. 2. High-Level On-Device Optimization Framework

fashion and the compiled code is uploaded to a central marketplace (e.g. Apple AppStore). This application is then downloaded onto numerous devices, each of which may vary in terms of hardware functionality and foundation library routines. This one-to-many relationship is ideal from a scalability and validation perspective, but has drawbacks in terms of possible compile-time optimizations. Since most mobile ecosystems are based on highly-flexible frameworks and foundation libraries, where much flexibility is built in to the framework in the form of dynamically-dispatched messages within the framework base classes, oftentimes only a limited portion of the framework code is used by a particular application.

Many compile-time solutions already exist to help reduce the overhead of dynamic dispatch, but require the ability to view the entire program space during compilation in order to make correct decisions [5], [6]. Unfortunately, applications that are downloaded onto the smartphone are pre-compiled binaries that have already been flattened. The ability to identify object-oriented class hierarchy information is extremely difficult, as the compiler-optimized assembly code will just consist of message sending using register values. To overcome this limitation, high-level class hierarchy and method information needs to be propagated onto the target device to enable a second-level of optimization to take into account the actual foundation code that is present on the device.

Furthermore, mobile processors often possess a large number of sensors and gadgets that can vary from device to device. For example, some devices may only have a GPS and accelerometer, while others also contain a compass, thermometer, barometer, etc. As the foundation libraries and applications often contain code to handle a wide range of sensors, for those devices that lack a given sensor the associated code is unnecessary. Pruning away such dead code reduces the quantity of polymorphic sites; removing unneeded method implementations reduces the possible targets for sending messages.

In order to accomplish this goal of improving mobile application performance, a novel on-device application code optimization framework is proposed. When applications are compiled using the mobile ecosystem toolchain, they are also analyzed in order to extract high-level application metadata, including class hierarchies and method implementations. This metadata is included in the application bundle that is uploaded onto the marketplace. Similarly, each new operating system release that is compiled will also analyze and annotate high-level class hierarchy and method implementation attributes, including the information in the OS system updates. Once an application is downloaded onto a given device, an on-device

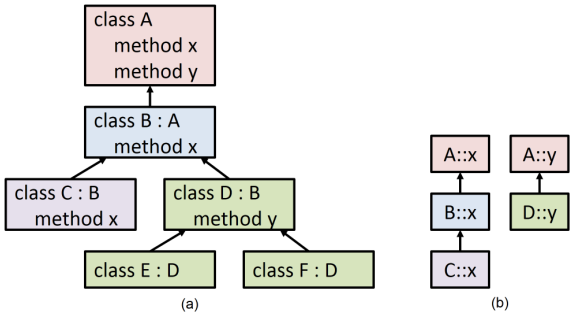

Fig. 3. Example Class Hierarchy and Method Partial Ordering

code analysis and transformation process will occur. Using the metadata from both the OS and the application, a complete object-oriented hierarchy view is available. Optimizations and code transformations are applied, also taking into account device specific attributes such as available peripherals and sensors. The result of this optimization will be a new application binary that can then be run on the device as would normally occur. A high-level overview of this framework is shown in Figure 2.

The key theory of this approach is the decomposition of the application's optimization process across multiple points in time. While many software optimizations can be achieved during regular compilation, the ability to do interprocedural optimization on mobile applications is problematic. The contents of operating system foundation libraries are not known for certain, so full application visibility is impaired during application compilation. This paper will show how a specific type of interprocedural optimization, *dynamic dispatch resolution*, can be accomplished by extracting partial class hierarchy information from multiple sources, and then employing a post-process code transformation on the application binary when all the necessary pieces of information become available. By widening the information channel from just pure binary instructions to also include class and method hierarchy metadata, powerful post-compilation code transformations can be performed.

The following subsections provide a detailed explanation on the steps involved in this framework.

A. Extracting Source Code Metadata

The first step of this framework is to capture high-level information related to the object-oriented classes and methods that would otherwise be unavailable by the time the application arrives on the mobile device. When a developer creates an application and compiles it for inclusion in the mobile marketplace, the critical information related to class hierarchies and message calls are captured. Similar to [5], the goal is to glean information about the possible receiver classes of each method being compiled.

Figure 3(a) shows a simple class hierarchy. Consider the case where the method *y* defined in class *D* sends the following message: `[self x]`. This would result in dynamic dispatch to determine which implementation of *x* to call based on the instance of `self` that exists at runtime. For example, the program may have instantiated class *E*, which inherits method *y* from class *D*, so the receiver of the message sent to `self` would be of type *E* at runtime and would need to determine which implementation of the method *x* to select. But, one may notice that no subclass of class *D* contains an overriding implementation of method *x*, and that the only possible implementation of method *x* in class *D* or any of its subclasses is the one defined in class *B*. Thus, the dynamic dispatch code of `[self x]` can be replaced with a direct method call to `B::x`.

At the time the application is compiled, information on the complete class hierarchy is not present due to much functionality coming from foundation code which can vary across devices. So, in order to allow this class hierarchy analysis to occur at a later time, the application compiler toolchain is augmented to extract the known class hierarchy information and store it as part of the application deliverable. In a similar vein, the same class hierarchy information is captured for the vendor-provided OS foundation library classes and stored for later reference. Once the application is downloaded onto the phone, the combined metadata from the application and OS will provide the complete view of all class hierarchies.

An additional stage of analysis is done for the foundation libraries to identify code that is needed for hardware-specific features. For example, there may be specific classes and methods that exist to interact with an ambient temperature sensor only present on some hardware devices. If the device lacks this sensor, the corresponding foundation code that interacts with and processes the sensor data is never used. By identifying these classes and methods, device-specific dead code can be eliminated when the OS is installed on the target device. The effort to annotate the foundation library source code with information about the physical properties required for the code's operation can be done centrally by the vendor and does not require any special effort from application developers.

B. Pruning Unused Classes and Methods

Once the particular application and foundation library are both present on a given physical device, pruning of unused classes and methods can be performed. As mentioned earlier, mobile foundation libraries are highly flexible and provide a myriad of framework base classes that may not all be utilized by an application. In order to reduce the sparseness of instruction code, unneeded library instruction code can be pruned away. Instead of having the application dynamically load the entire set of foundation libraries, only those libraries that are required by the application can be statically linked into the application binary. While the resulting binary will be relatively larger than the original application code, it is important to note that instruction code as a whole is quite small compared to other application data such as bitmaps, sound files, and databases, so the impact on overall storage space is trivial. On the other hand, the benefit is a much more compact instruction space that will reduce sparseness.

A straightforward analysis of the application's class hierarchy will provide a list of all possible classes that can be instantiated. This list is then used to identify any corresponding class definitions in the foundation library. These foundation classes are marked to be preserved. Then, a depth-first search is conducted on each preserved class to identify any required parent classes also defined in the foundation library that should be preserved. Lastly, the final set of preserved foundation classes that are uniquely required for the given application is statically linked into the application binary using hex-editing and address relocation where necessary.

Device-specific dead code in the foundation libraries is also removed whenever a new OS version is installed. A simple hex post-processing of the foundation code can remove unused classes/entry points for nonexistent hardware features.

TABLE II. DESCRIPTION OF BENCHMARK APPS

Bubbsy	Graphical world-based game (uses Cocos2D)
Canabalt	Popular run and jump game
DOOM Classic	3D first person shooter
Gorillas	Turn-based angle shooter (uses Cocos2D)
iLabyrinth	Puzzle navigation game (uses Cocos2D)
Molecules	3D molecule modeling and manipulation
Wikipedia	Online encyclopedia reader
Wolfenstein 3D	3D first person shooter

C. Optimizing Dynamic Dispatch Sites

To improve mobile processor application performance, as many polymorphic sites as possible should be removed. Many approaches exist to identify and replace resolvable dynamic message sending calls with static procedure calls. This paper relies on the same algorithm as defined in [5], in particular because that algorithm supports dynamically typed languages, such as Objective-C. The challenge for mobile processors is that applications are developed separately from the target device and foundation libraries, and require the additional metadata described earlier to enable post-processing of the compiled assembly code with visibility across the entire program space.

First, those classes that define new method implementations, versus those that simply inherit the implementation from their parent, are identified. A partial order of all methods is constructed, where one method M_1 is less than another method M_2 iff M_1 overrides M_2. Figure 3(b) shows the partial ordering corresponding to the example class hierarchy presented in Figure 3(a). An initial *applies-to* set is computed for each method containing the defining class and all its subclasses. Then, a top-down traversal of the partial method ordering is conducted, where each of the immediately overriding method's *applies-to* sets are subtracted from the current method's *applies-to* set. Thus, all methods will have an *applies-to* set that lists the set of classes for which that method is the appropriate target.

The application binary, including the statically linked foundation libraries, is now analyzed using a Perl script to try to identify polymorphic site optimizations. When a polymorphic call is encountered in the binary (i.e. an opcode calling the *_objc_msgSend* routine is detected by the Perl script), the metadata related to that call site is queried to get information on the receiver and selector. The class set that is inferred for the receiver is tested to see if any overlap exists in all the matching method *applies-to* sets. If no overlap exists, then the polymorphic site can be replaced by a static call since only one possible method implementation exists for that polymorphic site. In other words, the binary is hex-edited to replace the call to *_objc_msgSend* routine with a call opcode targeting the static address for the identified method implementation.

Furthermore, in the event that the static method implementation is very short (e.g. less than 15 instructions), the static call can then be replaced by inlining the target method's instructions. While normal optimizing compilers often employ code inlining for short routines, since the original code had dynamic dispatch the compiler would have been unable to resolve which routine to inline. Now that the dynamic dispatch site has been optimized using the additional information related to foundation libraries and the application's class hierarchies, a second pass for identifying inlining can be employed.

V. EXPERIMENTAL RESULTS

In order to assess the benefits from this proposed framework, an actual commercial mobile processor is utilized. The target mobile device chosen is an iPhone 4S (dual ARM Cortex-A9 800MHz processor, 512MB DRAM) running the iOS 5.1.1 operating system (Darwin Kernel 11.0.0). A set of eight open-source, interactive Objective-C iOS applications are used to benchmark the optimizations. A listing of these benchmarks and their respective descriptions is provided in Table II.

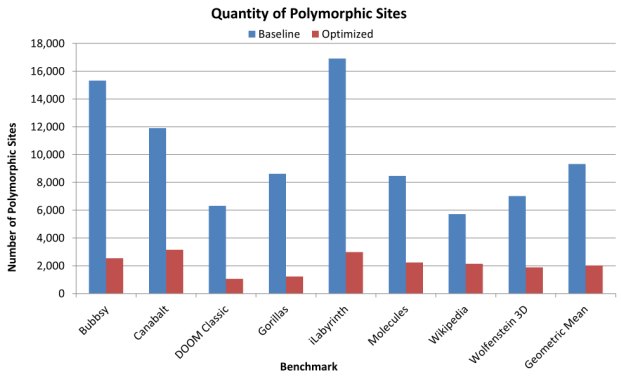

Fig. 4. Reduction in Polymorphic Sites

Each application is compiled using Xcode IDE version 4.1.1. One copy of the compiled application is the baseline without leveraging the proposed dynamic dispatch optimization framework, and another version contains the optimized binary code. Figure 4 shows a comparison of the quantity of polymorphic call sites between the two versions of a given application's binary code. A large majority of dynamic dispatch calls were able to be replaced by static method calls, resulting in a geometric mean reduction of 76.8% of polymorphic sites across all benchmarks.

In order to evaluate the corresponding performance improvement of reducing dynamic dispatch sites, both the baseline and optimized code will need to be executed on the physical device. This is necessary, as the vast majority of mobile applications are highly interactive and require touchscreen stimuli in order to realistically function. The selected benchmarks are no exception to this characteristic. By running the application directly on the phone and interacting with the application, the dynamic control flow will closely resemble typical usage. An additional benefit of running the application code directly on the smartphone is that the correctness of the code transformations are validated since they are executed directly on the processor.

The GNU *gcov* test coverage tool is leveraged to instrument executed code and to capture the dynamic execution frequencies of each line of source, as well as how many call and branch instructions occurred. In particular, both the settings of *Generate Test Coverage Files* and *Instrument Program Flow* are enabled in the Xcode compiler, and the *GCOV_PREFIX* and *GCOV_PREFIX_STRIP* environment variables are set to redirect the generated output profiles for all source files into a central path. As a result, when the application is executed on the phone, a corresponding *.gcda* coverage file is created for every source file. These coverage files are then downloaded from the phone and analyzed to determine the percentage of dynamic instructions that are control flow related.

To generate the dynamic instruction profiles for the benchmarks, both the baseline and optimized versions of the applications are launched on the phone and used normally for 5 minutes. As we are interactively using the application, the exact user input and resulting control flow will vary between every single run. In order to help reduce any outliers and noise, this process of running the application for 5 minutes is repeated 10 times to generate an arithmetic mean for each application.

TABLE III. TOTAL APPLICATION SIZE INCREASE COMPARED TO ORIGINAL BINARY CODE AND RESOURCE FILES

Bubbsy	Canabalt	DOOM Classic	Gorillas	iLabyrinth	Molecules	Wikipedia	Wolfenstein 3D	Geometric Mean
10.3%	1.9%	6.7%	5.5%	15.4%	8.0%	10.1%	17.5%	7.9%

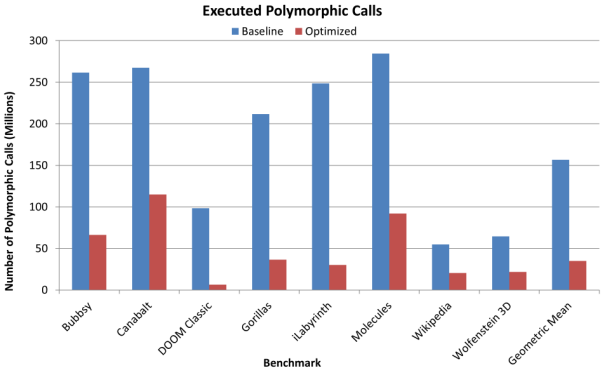

Fig. 5. Executed Polymorphic Calls

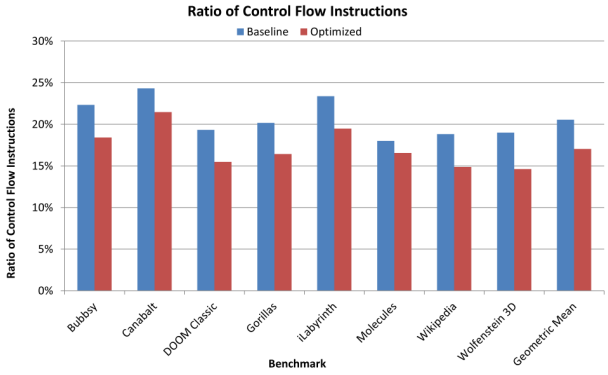

Fig. 6. Ratio of Control Flow Instructions

After collecting and analyzing all the *gcov* coverage files, the dynamic instruction count for polymorphic call sites is ascertained and shown in Figure 5. The geometric mean reduction of dynamically executed polymorphic calls across benchmarks is approximately 73.1%, which is quite significant.

As one can recall from Section III, every call to *_objc_msgSend* incurs at minimum 3 conditional branches (assuming the implementation is found in the first cache entry that is queried). If the method implementation requires further cache searching or a full class hierarchy search, the amount of control flow operations becomes significant. As polymorphic sites are replaced by static calls, the amount of control flow operations decreases. Furthermore, in the cases where static method inlining can be employed, all control flow is essentially obviated. Figure 6 shows the ratio of control flow instructions that are dynamically executed at runtime compared to other instructions, such as arithmetic or memory. The baseline proportion of control flow instructions has a geometric mean of 20.6%, whereas in the optimized case the mean is 17.0% illustrating the reduction in overall control flow instructions.

The code size overhead of creating the optimized version of an application is small. Applications usually include many other resource files in addition to the binary code, such as images, sound files, and config files. Applications typically consume 5–30MB of storage space on the device, of which the contribution of binary code is rather low. Nevertheless, since our proposal requires adding metadata used for post-processing and generating a transformed version of the binary code (while still preserving the original binary code), there is an increase to application size. Table III shows the increase for the total application size compared to the original binary code and resource files (the additional cost of metadata and transformed version of the code). Since most high-performance smartphones can store 16GB or more, this modest increase in size is acceptable.

Another important consideration for mobile processors is power. As shown, the proposed framework results in reductions to the number of control flow instructions that are executed. This has direct implications on power stemming from the branch predictor hardware. Branch predictors can contribute 10% or more of the total processor's power dissipation [12]. Reducing the number of conditional branches results in fewer branch predictor hardware queries, saving power.

VI. CONCLUSIONS

A novel framework for enabling on-device application optimization has been presented. Critical class and method hierarchy information is extracted during application compilation and conveyed along with the application binary when downloaded onto a device. The application is post-processed on the device, leveraging the complete knowledge of the underlying hardware features and OS foundation libraries that are present on the device. A device-specific optimized version of the application is created which greatly reduces the number of dynamic dispatch instructions. The benefits of such an approach have been demonstrated on real-world interactive mobile applications running on a commercial processor.

REFERENCES

[1] Apple Inc., "App store press release," Jan. 2013. [Online]. Available: http://www.apple.com/pr/library/2013/01/07App-Store-Tops-40-Billion-Downloads-with-Almost-Half-in-2012.html

[2] C. Zhang, H. Xu, S. Zhang, J. Zhao, and Y. Chen, "Frequency estimation of virtual call targets for object-oriented programs," in *Proc. of ECOOP*, 2011.

[3] A. Gutierrez, R. Dreslinski, T. Wenisch, T. Mudge, A. Saidi, C. Emmons, and N. Paver, "Full-system analysis and characterization of interactive smartphone applications," in *Proc. of IISWC*, 2011.

[4] D. Grove, J. Dean, C. Garrett, and C. Chambers, "Profile-guided receiver class prediction," in *Proc. of OOPSLA*, 1995.

[5] J. Dean, D. Grove, and C. Chambers, "Optimization of object-oriented programs using static class hierarchy analysis," in *Proc. of ECOOP*, 1995.

[6] O. Zendra, D. Colnet, and S. Collin, "Efficient dynamic dispatch without virtual function tables: the SmallEiffel compiler," in *Proc. of OOPSLA*, 1997.

[7] R. Teodorescu and R. Pandey, "Using JIT compilation and configurable runtime systems for efficient deployment of java programs on ubiquitous devices," in *Proc. of Ubicomp*, 2001.

[8] F. Pizlo, L. Ziarek, E. Blanton, P. Maj, and J. Vitek, "High-level programming of embedded hard real-time devices," in *Proc. of EuroSys*, 2010.

[9] Apple Inc., "objc class source code," Feb. 2010. [Online]. Available: http://www.opensource.apple.com/source/objc4/objc4-437/runtime/objc-class.m

[10] ——, "objc_msgSend source code," Feb. 2010. [Online]. Available: http://www.opensource.apple.com/source/objc4/objc4-437/runtime/Messengers.subproj/objc-msg-arm.s

[11] J. Joao, O. Mutlu, H. Kim, R. Agarwal, and Y. Patt, "Improving the performance of object-oriented languages with dynamic predication of indirect jumps," in *Proc. of ASPLOS*, 2008.

[12] D. Parikh, K. Skadron, Y. Zhang, M. Barcella, and M. Stan, "Power issues related to branch prediction," in *Proc. of HPCA*, 2002.