3.1 Feature phone in Developing Countries

Developing countries have seen a massive explosion of mobile phones in the past decade. As per the Telecom Regulatory Authority of India (TRAI), there are nearly 800 million cellular subscriptions in India alone [46]. Mobile phones have been used for various purposes in developing regions such as facilitating education [47, 48, 49], healthcare delivery [50, 51], agriculture information dissemination [52], and crowdsourcing [53].

With recent trends showing that the usage of mobile internet has surpassed desktop internet in India [54]. Developing countries have a huge population of feature phone users. A recent study [56] brings out that as much as 94% of all phones shipped in the first quarter of 2012 in India were feature phones, and nearly 50% will remain so 5 years from now. The percentage number of feature phone users will be even higher in the underserved sections of the society who are the primary target audience for the ICT4D community. The complimentary problem of improving internet/web access in the developing world has been addressed by several works. Some examples include, TinyPC[57], accelerating web access using a caching and prefetching engine [58], RuralCafe [59] and mango [59].

Looking at the day by day demands of web page in feature phone, there is lot of necessity to optimize the JavaScript engine in respect to both the size of the code and the algorithms that are used for interpreting script. In this research work we propose a compiler and interpreter architecture of JavaScript Engine for the feature phone. Typically, feature-rich phone manufacturers now use 8 to 16 Mb of low-power SRAM for data backup; 32 to 128 Mb of Pseudo-SRAM (PSRAM) for the working area of the system; and 64 to 128 Mb of NOR Flash for bootable code storage on basic phone programs[60]. These devices perform Internet browsing, text messaging, games, the downloading and playing of music, and digital-camera functionality and applications (including the ability to take, transmit, receive, and display photos). Such applications have caused an increase in the complexity of memory requirements.

To implement JavaScript Engine into a feature phone we need to consider execution time, memory and battery constrains of the feature phone and the dynamic behavior of JavaScript [61].
3.2 JavaScript Optimization, Point to Analysis and Function Pointers

In Designing and implementing JavaScript interpreter for wireless embedded systems is a challenging task. Sound Compilation and Interpretation concepts, techniques and implementations have been developed many years back and are around till date. One of the main areas that vastly explored is the inter-mediate representation of the source code. This representation varied from simple AST's, CST's, byte-code, machine instructions etc. Any representation form that is chosen is based on the requirements and constraints of the language and platforms. Most of these representations are developed keeping in mind the desktop computer which has sufficient computing resources like memory, CPU and disk space with other OS capabilities like multi-threading and multi-tasking. Except for KVM there is no other known implementation that works satisfactorily for the smaller electronic devices. In object oriented programming language like C++, methods of the class are being called by the instance of the class (object) through the function pointer. A function pointer is a variable that stores the address of a function that can later be called through that function pointer. This is useful because functions encapsulate behavior. For instance, every time you need a particular behavior instead of writing out a bunch of code, all we need to do is call the function [62].

For any application, function pointer is used for faster execution; if the numbers of function-pointers are huge, it consumes lot of initial memory during the starting of the application, which could be the bottleneck for the small mobile devices. Also there are some disadvantages of function pointer. From “Context-Sensitive Inter-procedural Points-to Analysis “When a function is called through a function pointer, program execution could potentially jump to a number of functions, which we cannot know for sure ahead of time” [63]. "Everything" in JavaScript is an Object: a String, a Number, an Array or a Function. All the objects and their properties are being loaded into the environment before compilation of any script. Depending on the script, these properties are invoked at the runtime through the function pointers. For reducing the high peak memory, we redefine the object structure.

A lot of research work has been carried out in the field of optimization of Java Script. Authors in [64] have proposed a small-step operational semantics for the ECMA-262 Script standard language corresponding to JavaScript, which could ultimately establish equivalence among some JavaScript programs. The effect of a thread level speculation approach is evaluated by authors in [65] on top of Rhinol 7R2 JavaScript engine using JavaScript
benchmark suite V8. Further, the effects of null return value prediction approach for function calls as well as conflicts with variables in a global scope are measured too. Although the strategy of speculation on return values is established to be successful, and for some applications, the execution time is bettered, still the speculation overhead leads to performance degradation in case of some applications. An empirical study on prevalence of privacy violating information flows on a large number of popular web sites is presented in [66]. Authors have extended the study over the Alexa global top 500,000 websites of four privacy violating flows: cookie stealing, location hijacking, history sniffing and behavior tracking. The dynamic behavior of a number of widely-used JavaScript programs are analyzed in [67] and the justification and method of implementation of dynamic features into JavaScript programs are successfully established. Authors in [68] present an elegant syntax for object oriented programming in JavaScript, which provides a solid fundament for JavaScript programmers to understand the JavaScript object system. A hybrid type inference algorithm based on points-to-points analysis is proposed [69], which could successfully contribute to optimization of JavaScript behavior. Incremental static analysis as a way to analyze streaming JavaScript programs is explored [70], where use of combined offline-online static analysis to accomplish fast, online incremental analysis at the expense of costly offline analysis on static code is demonstrated. A points-to analysis for JavaScript is presented by authors in [71], which incorporates an analysis of points-to behavior of JavaScript followed by an evaluation with the application of the analysis to a code optimization technique to establish the usefulness of points-to analysis. A polymorphic type inference algorithm for a small subset of JavaScript is proposed in [72] in order to prevent accessing undefined members of objects, where a type system is defined that allows explicit extension of objects through add operation implicit extension through method calls. A Rapid Atomic Type Analysis (RATA) by abstract interpretation, an application to JavaScript optimization is presented in [73], which represents a static analysis based on abstract interpretation for the rapid inference of atomic types in JavaScript programs.

Several things are thought to be correlated with optimization in JavaScript engine. Some optimization includes static analysis through pointer analysis [74]. Precise information about memory accesses and procedure calls is fundamental for a large number of static analyses used in optimizing compilers and software engineering tools. The goal of pointer analysis is to determine the set of memory locations that a given memory location may point to. In addition, pointer analysis determines which function addresses may be stored in a
given function pointer. Because of the importance of such points-to information, a variety of analyses have been developed [75-93].

Though function pointers increase the runtime execution, there are many disadvantages of function pointer has been discussed. To address security the Point-Guard, a compiler technique to defend against most kinds of buffer overflows by encrypting pointers when stored in memory, and decrypting them only when loaded into CPU registers [94]. Function pointer validator, a new solution capable of dynamically validating the type equivalence between function pointers and target functions, which can detect all function entry attacks that violate type equivalence [95].

There is no research has been done on static memory occupied by the function pointers. In JavaScript Engine there are 138 function objects, which all are invoked by the web-browser at runtime through function pointer, which interim occupies a huge static memory of the mobile device. Looking to the security and the memories we propose one optimize solution which will save lot of static memories of the mobile device.

3.3 Compiler, Interpreter and AST node Optimization

For many embedded applications, program code size is a critical design factor. Many techniques have been done to reduce the code size. One promising approach for reducing code size is to employ a “dual instruction set”, where processor architectures support a normal (usually 32-bit) Instruction Set, and a narrow, space-efficient (usually 16-bit) [96]. Advances in semiconductor technology permit increasingly complex applications to be realized using programmable systems-on-chips (SOCs) [97,98]. Several embedded processors have load- and store-multiple instructions, representing several loads (or stores) as one instruction [99].

Many researches have been done for compiler optimization. Authors in [100] propose an Architecture Description Language (ADL)-driven method for accurately capturing a wide range of programmable architectures and thereby generating efficient compilers. A compilation framework for code reduction in embedded applications presented in [101] uses a heuristic, leading to a dual instruction set which is space-efficient. A compiler optimization problem is addressed to enhance energy efficiency of Register File (RF) protection with respect to a number of embedded applications [102]. A dynamic code mapping technique is presented by authors in [103], which uses a heuristic algorithm that
creates an efficient Function-to Space mapping with a given code size in the local storage, thereby increasing the speedup. A compiler technique based on inter-procedural code analysis for reduction of vulnerability of RF to soft errors is proposed in [104], which uses a heuristic algorithm in order to optimize overhead reduction, consequently leading to an effective reduction of code size overhead. Simultaneous Determination of Regions and Function-to-Region Mapping for scratchpad memories (SRDM), a fully automated, dynamic, code overlaying technique based on pure compiler analysis for energy reduction for on-chip scratchpad memories in embedded systems, is detailed in [105], which is capable of reducing total energy consumption, consequently reducing the energy of the memory subsystem as a whole. Authors in [106] present a Compiler-in-the-Loop (CIL) framework for Design Space Exploration (DSE) processor architecture that comprises of an optimizing compiler, an instruction set simulator and a cycle-accurate simulator for earlier estimation of performance, power and code size. A customizable retargetable compiler framework is presented in [107] that is capable of determining phase-ordering between different transformations dynamically as per the resource availability and program region characteristics. Here, authors conduct experiment with ordering If-Conversion, a predicted execution technique and Speculative code motion, and the results demonstrate the flexibility of ordering of the transformations while compiling. The compiler approach to optimization of memory hierarchy along with a set of challenges to it is addressed by the authors in [108]. Authors in [109] focus on two tasks to instruction selection phase: first is to find an efficient algorithm which would generate an optimal instruction sequence and second is the automatic generation of instruction selection programs.

Here, we are purposing compiler for feature phone. Implementing an abstract syntax tree (AST) interpreter is easy for a new programming language. Creating AST feels natural as it parsers for context-free grammars process source code in a tree-like fashion. However it is costly for interpretation and difficult to optimize, as AST interpreters is calling the execute method for every node requires, and execute methods are highly polymorphic. Due to this reason, language implementers define bytecodes to speed up interpretation. It is well understood, and many variants with varying dispatch costs have been studied [110, 111]. Bytecodes are set of virtual instruction, and can be defined freely without worrying about hardware constraints. Two different strategies are defined for bytecodes one is language-independent bytecode and another is language specific bytecodes while former is used internally by VM latter aim to provide an instruction set that is suitable for many languages. JavaTM bytecodes [112], originally designed primarily for the Java programming language,
are now used for many languages. However, byte codes are difficult to modify once they are emitted. For example, to replace one generic byte code (a call to a type-agnostic addition method) with a short sequence of type specialized byte codes (a few type guards followed by an integer addition bytecode) requires adjusting offsets in branch instructions that jump over the replaced code.

### 3.4 Selection of Interpreter

Converting from AST to byte code is done in two passes. The first creates the namespace (variables can be classified as local, free/cell for closures, or global). With that done, the second pass essentially flattens the CFG into a list and calculates jump offsets for final output of byte code. It has been observed that in JavaScript many of the scripts are compiled but not executed at runtime. It is an overhead to the compiler and will consume a lot memory and power of the embedded devices. Looking at this in our JavaScript engine we make an AST based interpreter rather than popular byte code based. Appendix-II shows the comparisons of byte code against the AST node.

By optimizing AST node we can achieve better result than byte code interpreter. One classical example is rewriting of AST node [113]. For example, a tree node for a type-agnostic addition can be replaced with a specialized integer addition node after the first execution of the addition reveals that the operands were integers. This is based on the observation that types are mostly stable, even in dynamically typed languages. While naive interpretation of an AST is slower compared to byte codes, use of profiling and AST rewriting allow an AST interpreter to achieve speeds comparable to the speed achievable with language-independent byte codes [113].

In this research work we defined many AST node optimization techniques by targeting frequently used parameters. In JavaScript, identifiers, If-then-else, dot operators and function call are used frequently and it has been observed that these parameters are consuming considerable amount of static memory of device. We improve the AST building logic to overcome the same and provide an optimization solution.

Interpreting the AST node is a big challenge in a constraint memory like feature phone. While traversing the AST node either recursive or iterative method can be chosen. Recursion is often used without considering any alternatives and it is true that recursive solution is often more elegant and easier to implement than the iterative solution. Moreover,
iterative solutions are usually more efficient than recursive solutions as they don't have the overhead of the multiple method calls.

When we have to perform a complex task we need to use recursion that can be broken into the several subtasks. Recursion is implemented as a method that calls itself to solve subtasks. During this process recursive call the values of the local fields of the method are placed on the method stack until the subtask performed by a recursive call is completed. Thus, whenever recursive method is invoked, local fields are put on the method stack and used again after the recursive call is completed. Sometimes a recursive solution doesn't preserve any local fields during the recursive call, is known as tail recursions. Tail recursions are recursions where the recursive call is the last line in the method. William D Clinger, in “Proper Tail Recursion and Space Efficiency’, proper tail recursion concerns the efficiency of procedure calls that occur within a tail context. When he examined closely, proper tail recursion also depends upon the fact that garbage collection can be asymptotically more space-efficient than Algol-like stack allocation. Proper tail recursion is not the same as ad hoc tail call optimization in stack-based languages. Proper tail recursion often precludes stack allocation of variables, but yields a well-defined asymptotic space complexity that can be relied upon by portable programs. Tail recursions are generally considered a bad practice [114, 115] and should be replaced with iteration. In the November, 1963 issue of communication, James A. Ayer leaves unanswered the truth of the claim that" all recursive relations can be reduced to recurrence or iterative relations". After that many research has been done to convert recursive function call to iterative method [116, 117].

Ariel O. (2008) presents S-expression Interpreter Framework (SIF) based on the interpreter design pattern and written in the Ruby programming language in order for language design and implementation, which can be used for demonstration of advanced language concepts and various programming styles [118]. A comparison of two versions of an interpreter for Java programming language is performed in the paper of Mark H. and others (2011), where the authors chose the versions such as visitor pattern and interpreter pattern and the comparison is carried out with respect to maintenance and execution efficiency of implementation of Java programming language[119]. Design of an interpreter with a virtual hardware management facility is detailed by Oliver D. et al (2002), which overcomes the Field-programmable Gate Array (FPGA) resource limitations and enables implementation of large systems with small FPGA chips [120] Design and implementation of a query language interpreter with object oriented specification for bibliographic
information retrieval is presented by Igor F. and others (1999) that uses an internet client application in Java programming language [121]. Effect of mis-predictions during execution of the indirect branch instructions on an interpreter is addressed by Wien T.U. (2003) [122]. Effects of “recursive make” related to UNIX related programs are discussed by Peter M. (2008) [123]. Robert H.S. et al (1960) provide a synthesis of recursive vs. non-recursive systems with respect to interpretability of a parameter [124]. TyCL (Typed Command Language, an implementation of the Tcl language that is aimed at producing better results during compilation, is presented by Andres B. (2011) [125]. A debuggable interpreter design pattern is included in the work of Jan V. (2009) that specifies the coexistence of multiple debuggers in order to accept new debugging operations, and at the same time being easy to implement [126]. The calculational design of a generic abstract interpreter for a simple imperative language is detailed by Patrick C. (1999) [127].

Though we choose iterative method for interpreting the JavaScriprt Engine next challenge is to decide whether to select iterative thread based or iterative stack based. Though thread based fast, occupies less ROM size however at compilation time it allocates lot of memory which would be a bottle neck for the small feature phone. Looking to the memory constraint we choose iterative stack based interpreter or non-recursive stack engine to interpret the JavaScript.

3.5 The closure property and Runtime Stack

JavaScript is an imperative, object oriented language with Java-like syntax, but unlike Java it employs a prototype-based object system. An object is a set of properties, a mutable map from strings to values. A property that evaluates to a closure and is called using the context of its parent object. The objective of the closure is to keep track of objects created during the execution of a method. The objects may be created directly in the method or in methods called by the method. An object is considered to have closure from the scope of a method if a reference to the object is returned from the method, or if a reference to the object is assigned to a field of an object. While keeping the parent reference a subsequent memory leaks will be happened during the execution as a result.

Numerous works have been proposed on Run-time systems in the literature by various authors. Authors in [128] use a Distributed Cactus stack in order for Runtime stack sharing in the Chime Parallel Processing system that enables real shared-memory multiprocessor semantics, which is achieved by a runtime system. In addition, the runtime system is also able to provide nested parallelism, task synchronization, load balancing as
well as fault tolerance. Using closure conversion, an attempt is made by the authors in [129] to bridge the gap between the functional evaluators and abstract machines for the λ-calculus. Reduction of run time of indirect branch statements is addressed in [130]. A stack based platform independent Smart Virtual Machine (SVM) is detailed in [131] that is capable of running on different smart devices. An analysis of runtime stacks manipulation by various programs on Intel x86 machines is discussed in [132], which optimally improves reasoning and manipulation of values stored on the run time stack. In order to bridge the gap of speed between the processor and memory a stack cache memory is proposed in [133], which demonstrates a hit ratio of 99%. A flexible Java execution environment for mobile devices is proposed in [134], which supports dynamic adaptation to needs of the applications and available resources. A JavaScript Abstract Machine (JAM), detailed in [135] represents a precise model of stack structure and it confers precise control-flow analysis in the presence of control effects such as exceptions. Two scripting languages, namely, Yolan and LightScript proposed in [136] enable scripting on low-end mobile devices and are capable of loading and executing scripts in the source form at run time with a run time stack. Authors in [137] present a modular design of JavaScript, where the modules can be loaded from external sources and it supports a flexible mechanism for dynamically loading the code isolated from untrusted modules. An integrated compile-time and run-time system is described in [138] which contributes to enhanced efficiency of shared memory parallel programming on distributed memory systems. Run time fault location and its isolation in a multi-language program can be successfully addressed by the debugging tool proposed in [139] with a CallStack. A run time system discussed in [140] is capable of providing a variety of run time facilities such as easy implementation of code generators, quick program execution and efficient garbage collection. The cactus-stack problem is addressed in [141] using Thread-local Memory Mapping (TLLM) for re-implementation of cactus-stack in the open-source Click-5 runtime system. A symbolic stack addressing can help for fetching and storing without significantly increasing execution time, as detailed in [142]. An exclusive study on verification and analysis of symbol table for C++ compiler is described in [143].

In a complete interpreter, the runtime stack plays a much greater role. At any point during the execution of a program, the stack represents a “snapshot” of the execution state. It contains the information about the procedures and functions that have been called, their return values and addresses, the current values of all their formal parameters and local variables, and which of these values are accessible according to the scope rules [144]. We develop a closure algorithm using runtime stack.
With the paradigm shift of applications from desktop to hand held devices, the number of mobile operating systems or platforms emerging is enormous. Since each platform has its own architecture and interfaces, developing applications for multiple platforms is a tedious and daunting task for any mobile application developer. A framework for developing cross platform applications has become the need of the hour [145]. Mobile devices run on diverse platforms requiring differing constraints that the developer must adhere to. Thus, extra time and resources must be expended to develop multiple versions of a single application for the different platforms. There have been attempts to minimize the need for these extra costs with mobile cross platform development environments such as Titanium, PhoneGap, and Corona. They are relatively new to the mobile application building world, and though they have the same goal, their approaches are quite different [146]. Cross-Platform mobile development tools are gaining popularity in the world due to their characteristic to compile the application source code for multiple supported OS’s. Such tools are mainly depending on web programming languages like HyperText Markup Language (HTML), JavaScript and Cascading Style Sheets (CSS) with some native wrapper code for accessing native Application Program Interfaces (API) like Camera, Contacts, etc. The application development is very easy and time saving with these tools [147].

Each platform allows us to instantiate a browser instance, chromeless, and interact with its JavaScript interface from native code. From within that Webview we can call native code from JavaScript. This is the hack that became known as the PhoneGap technique pioneered by Eric Oesterle, Rob Ellis, and Brock Whitten for the first iPhone OS SDK at iPhone dev Camp in 2008. This approach was later ported to Android, BlackBerry, and then to the rest of the platforms PhoneGap supports. PhoneGap is an open source framework that provides developers with an environment where they can create apps in HTML, CSS, and JavaScript and still call native device features and sensors via a common JS API. The PhoneGap framework contains the native-code pieces to interact with the underlying operating system and pass information back to the JavaScript app running in the Webview container [148].

Future embedded and ubiquitous computing systems will operate continuously on mobile devices, such as smartphones, with limited processing capabilities, memory, and power. A critical aspect of developing future applications for mobile devices will be ensuring that the application provides sufficient performance while maximizing battery life. Determining how a software architecture will affect power consumption is hard because the
impact of software design on power consumption is not well understood. Typically, the power consumption of a mobile software architecture can only be determined after the architecture is implemented, which is late in the development cycle when design changes are costly. Model-driven Engineering (MDE) is a promising solution to this problem. In an MDE process, a model of the software architecture can be built and analyzed early in the design cycle to identify key characteristics, such as power consumption [149]. Authors [150] propose AIOLOS, a mobile middleware framework for improving mobile application performance through cyber foraging on the Android platform. AIOLOS uses an estimation model that takes into account server resources and network state to decide at runtime whether or not a method call should be offloaded.