Optimizing AST Node for JavaScript Compiler A lightweight Interpreter for Embedded Device

Sambit Kumar Patra
COMVIVA Technologies, Bengaluru, India
Email: sambit.para@comviva.com

Binod Kumar Pattanayak
Institute of Technical education and Research,
Siksha 'O' Anusandhan Deemed to be University, Bhubaneswar, India
Email: bkp_iter@yahoo.co.in

Bhagabat Puthal
Institute of Technical education and Research,
Siksha 'O' Anusandhan Deemed to be University, Bhubaneswar, India
Email: bputhal@gmail.com

***Abstract*— Limited memory in mobile devices presents the bottleneck to execution speed. Abstract Syntax Tree (AST) could be a better option for mobile codes as it is compiled only once. Hence, to improve the execution speed of a mobile device, AST node should be designed in such a way that it would consume less memory and improve the execution speed. In this paper, we implement AST node optimization technique for a set of frequently used operators and statements on a JavaScript compiler, such as Identifiers, Function-calls, Dot-operator and If-then-else. The desired optimization is justified by experimental results.**

***Index Terms*— AST node, JavaScript Compiler, Optimization**

I. Introduction

In the Script Engine architecture, the compiler component generates the AST and symbol tables. The interpreter executes the AST tree with reference to the symbol table. The other possible alternative is that the compiler generates the byte-code. Traditional byte-code generation involves 2 stages of compilation. At first, it generates AST and then byte-code from the AST. Many times, it has been observed that the scripts are compiled but not executed. Considering the memory limitation of the mobile devices and the limitation of execution, it is preferable to generate AST node and execute as and when required rather than converting all AST nodes to respective byte-codes. The Script Compiler converts the script code into an equivalent representation called Abstract Syntax Tree (AST). The AST is a structured tree format of the source script containing different nodes, which represent different constructs of the language and the references to the symbol table entries. The process of creation of AST involves interactions with the Symbol Table Manager for creating Symbol Table and updating information in it. AST and Symbol Table are used by script interpreter to perform evaluation. AST forms an end result of the parser after evaluation of the grammar rules.

In this paper, we have addressed the problem of AST node optimization for a set of frequently used statements on a JavaScript compiler. We have ported, tested and verified our script engine with devices a) Moto RAZR v3 (brew 3.15), b) Qtopia (Linux OS), c) Samsung (Windows), and d) Nokia Series (Symbian OS). We have implemented a JavaScript engine as per the ECMA (European Computer Manufacturers Association)-Script specification and then integrated the same with a web browser. The performance and memory consumption of AST nodes for different operators and statements like identifier, dot-operator, if-then-else and function-call are observed. The memory consumption pattern of this structure with various Alexa sites have been taken into consideration and we have come up with another alternative structure, which if used, shall improve the memory consumption drastically. During implementation, we have attempted to optimize the size of the structure and have observed the improved performance of the Alexa top 10 sites. In this paper, we intend to discuss the problem of memory consumption of the AST structures for identifier, function call, dot-operator, if-then-else and devise solutions to reduce the memory occupied by them. In the AST node, identifiers are represented as unary nodes; function calls and dot operators as binary nodes and if-then-else as ternary node.

II. Related Work

Authors in [1] 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 [2] uses a heuristic, leading to a dual instruction set which

doi:10.4304/jcp.8.2.349-355

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 [3]. A dynamic code mapping technique is presented by authors in [4], 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 [5], 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 [6], which is capable of reducing total energy consumption, consequently reducing the energy of the memory subsystem as a whole. Authors in [7] 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 retarget able compiler framework is presented in [8] 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 [9]. Authors in [10] 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.

III. AST NODE AND REPRESENTATION

A AST Node

The Script Compiler converts the script code into an equivalent representation called Abstract Syntax Tree (AST). The AST is a structured tree format of the source script containing different nodes that represent different constructs of the language. AST forms an end result of the parser after evaluation of the grammar rules.

The syntax analyzer creates the Abstract Syntax tree (AST) using bottom-up approach. This approach involves creation of leaves of the tree and then the created nodes are linked to form a structured tree. An AST node represents the type of operation that is allowed on its children and the structure of the children. Depending on the operation supported, the AST node is categorized as unary, binary or ternary. In this paper, we mainly focus on optimization of AST node for an identifier (unary node), dot operator (binary node), function call (binary node) and if-then-else (ternary node) statements. The following is the structure of a typical AST node.

```
typedef struct _ast
{
        E_TYPE  eNodeType ;
         void *pBranch ;
} AST;
```

Here, eNodeType specifies the node type, where as pBranch is a pointer which contains the data of that node type.

For an identifier, the eNodeType is “E_IDENTIFIER_NODE” and the pBranch points to the structure of an identifier. An identifier structure contains the symbol reference and the symbol table entries.

Example of binary: For a function call, the eNodtype is “E_FUNCTION_CALL” and the pBranch points to the structure of a binary node, function. Function name and the arguments are the two parameters for the function call. To represent the AST node for the function call, the function name is in left hand site of the AST node which is an identifier node and arguments are in right hand site of the AST node which is the identifier list node.

For a dot operator, the eNodeType is “E_DOT_TOK” or “E_DOT_TOK_LIST” and the pBranch points to the structure of a binary node. Object name and its property name are the two parameters for the dot operator. AST node for dot operator is represented as an identifier appearing to the left of the dot operator which is the object and another identifier appearing to its right which is the property of that object. Thus, to specify the property of an object, a dot operator is used in between two identifiers, i.e. a.b, where “a” is an object and “b” is its property. In order to specify the property of an object, which is a property of another objet, multiple dots can be used between the identifiers, i.e. a.b.c, where “c’ is a property of object “b” and “b” is a property of object “a”. For single dot operator, eNodeType is “E_DOT_TOK”, and for multiple nodes, eNodeType is “E_DOT_TOK_LIST”. Similarly other binary nodes are conditional statement, mathematical statements (+, -, *, /) etc.

Example of Ternary: For if-then-else, the eNodeType is “E_IF_THEN_ELSE” and the pBranch points to the structure of a ternary node, if-then-else. Conditional statement, true statements and the false statements are the parameter of if-then-else. To represent the AST node for the if-then-else, the conditional statements are in child-1 of the AST node which are binary nodes or list of binary nodes, the true and false statements are child-2 and child-3 of the AST node which are either unary, binary, ternary or the list of those nodes.

B. Identifier node, Binary Node and Ternary Node

A variable is an identifier**,** whose name is stored in the string table and its value in the symbol table. The node representing the identifier contains pointer to the entry in the string table as well as a reference to the symbol table entry. The process of creation of AST for an identifier involves interactions with the string table for getting the

TABLE I.
AST REPRESENTATION

Operator	Representation	
	Category	Structure Description
Identifier	Identifier	vNameRef- String table entry vRefSymTab - Symbol table entry
Function call	Binary	pLeft – identifier or expression node pRight – arguments (Argument List)
Dot operator	Binary	pLeft – Identifier pRight – Identifier
If-then-else	Ternary	pC1 – conditional expression node pC2 – list of then part statements pC3 – list of else part statements

symbol reference and symbol table entries for updating information in it.

```
typedef struct _Identifier
{
        void *vNameRef;
        void *vRefSymTab;
}IDENTIFIER; (Size: 8 bytes)
```

The symbol table provides an interface to store the data pertaining to a symbol and whenever a lookup is required an appropriate reference to the symbol table entry is returned back to the caller. It is used to reference the global scope by default and whenever the control enters a new scope or local scope logically (i.e when function is encountered), the symbol table is updated with the current information of the function name, the reference of the function points to a newly created scope. The handle is assigned with the new scope reference. The symbol table is responsible for managing the storage of the symbols encountered during the process of compilation and interpretation of the scripts.

Other categories include Binary and Ternary, which will contain pointers to one, two and three separate AST nodes respectively, as given below:

```
typedef struct  _Binary
 {
        AST *pLeft;
        AST *pRight;
}BINARY;  (Size:8bytes)
typedef struct _Ternary
{
        AST *pC1;
        AST *pC2;
        AST *pC3;
}TERNARY; (Size:12bytes)
```

The function call and dot operators are represented with binary nodes while if-then-else operator with ternary node.

IV. PROBLEM DESCRIPTION

The Script Compiler contains a unit called the Lexical Analyzer (LA) that interacts with the Syntax Analyzer or parser and provides the tokens with reference to symbol table when requested. The LA generates tokens for input buffer provided by the parser. Yacc tool parses the input grammar and generates source code for parsing.

TABLE II
STRUCTURE SIZE CALCULATION

Operator	Memory Size			
	Structure Description		Size in Bytes	Total Size in Bytes
AST	eNodeType		4	8
	pBranch		4	
Identifier	E_IDENTIFIER_TYPE		4	12
	pBranch	vNameRef	4	
		vRefSymTab	4	
Function Call	E_FUN_CALL_TYPE		4	28
	Identifier (Function name)		12	
	Identifier (1 parameter)		12	
DOT Operator	E_DOT_TOK		4	24
	pLeft -> Idetifier		12	
	pRight ->Idetifier		12	
If-then-else	E_IF_THEN_ELSE_TYPE		4	16
	pC1-> conditional expression node		4	
	pC2–> then part statements		4	
	pC3–>else part statements		4	

A. Identifier Node

In java script, when a variable is declared with the keyword ”var”, then script compiler creates an identifier node and resolves it by creating a symbol table entry either in the global scope or in the local scope depending on the context of declaration. If it is declared inside a function, then it creates a symbol table entry in local scope else in global scope. Many times the variables are not declared with “var” keyword or global variables are used inside the local function and these are resolved at the time of execution. These are known as unresolved identifier nodes. In order to improve the speed of the execution, it is required to reduce the memory of the identifier node thereby optimizing the structure of the node. Currently, for a single identifier node, 12 bytes are used (Table II). We can reduce the size of identifier node to 8 bytes.

During Compilation, once Lex gets the identifier token, it searches the string reference from the string table; Yacc gets symbol table reference from symbol table and generates an identifier node with string reference and symbol table reference. If particular symbol is not added to the symbol table in that scope (global or local), the symbol table reference is NULL and known as unresolved identifier. The unresolved identifier would be

resolved at the time of interpretation by replacing string table reference with symbol table entry. As many times the variables are in unresolved state, the unused NULL pointers are huge in number [Table III, Fig.1].

TABLE III
STATISTICS OF REPORT OF UNRESOLVED IDENTIFIER NODE

Alexa Sites	Unresolved Identifier Node		
	Global Scope	Local Scope	Total Unresolved Identifier
CNN	439	2328	2767
MSN	67	6388	6455
EBAY	612	18	630
AMAZON	56	3253	3309
REDIFF	57	175	232
NDTV	105	2559	2664
YATRA	165	2002	2167
ALIBABA	317	4530	4847
APPLE	1209	3421	4630

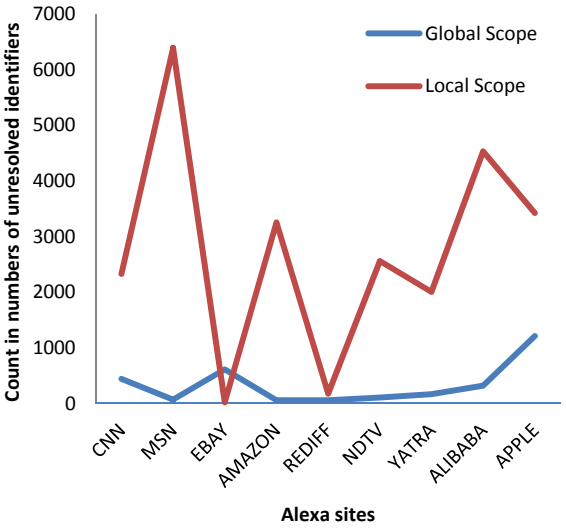

Figure 1. Statistics of memory report for unresolved identifier node

B. Dot Operator Node

The Dot operator feature is used widely in java script. During Compilation, once Lex gets the dot operator, it returns TOK_DOT and the next token will be an identifier. Then Lex looks up the string reference of the Identifier from the string table. Yacc creates an Identifier node using string table reference only without the Symbol Table reference. The identifier node present on the right hand side of the DOT operator is used for lookup from the Object using the string reference only and not the symbol table reference during Interpretation. Hence the symbol table reference which is not used can be removed for optimization.

Script interpreter interprets the dot operator from right to left unlike binary operator. It starts evaluation from left which is an object and then the property. To make it easy for interpretation, it is required to restructure the dot operator.

C. Function Call and If-then-else Node

Function calls and if-then-else are also frequently used in java script. Minimum 28 bytes are being used for a function call with a single argument and 16 bytes are used for if-then- else node (*Table II*). In order to improve the speed of execution as well as to increase the speed of script engine, it is required to reduce the memory of the function call node and if-then-else node. Very often, the functions are called without the arguments and if-then statement is used without else [Table IV, Fig.2].

TABLE IV
STATISTICS OF REPORT OF DOT OPERATOR, IF-WITHOUT-ELSE, FUNCTION CALL WITHOUT ARGUMENTS

Alexa Sites	Dot operator, Without Else and Function call without Arguments		
	Dot operator	Without Else	Function call Without Argument
CNN	5086	604	71
MSN	4416	859	256
EBAY	6080	453	18
AMAZON	1978	1075	58
REDIFF	113	30	2
NDTV	945	936	126
YATRA	5041	524	26
ALIBABA	5583	758	67
APPLE	8274	1019	27

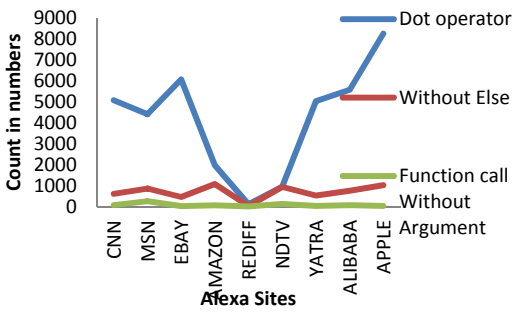

Figure 2. Statistics of memory report for Dot operator, if-without-else and function call without arguments

V. Proposed Solutions

A. IdentifierNode

If the identifier is already resolved, it is not required to keep the string table reference as it is also available in the Symbol table reference. Similarly, for the identifier, which is not resolved, the symbol table reference is not required as it would be resolved during interpretation. As shown in Table III, many unresolved identifiers are being used in the web page and their symbol table references are NULL. In order for optimization, the NULL pointer, 4 bytes in size, can be removed from the identifier node.

In this proposed solution, there are two node types instead of one node type (E_ATYPE_IDENTIFER). These are:

E_ATYPE_IDENTIFER_RESOLVED: If the identifier is in the symbol table, then the node type is E_ATYPE_IDENTIFER_RESOLVED and it will contain the address of the symbol table reference only.

E_ATYPE_IDENTIFER_NOTRESOLVED: If the identifier is not in the symbol table, then the node type is E_ATYPE_IDENTIFER_NOTRESOLVED and it will contain the address of the string table reference only. During the interpretation, interpreter will resolve that identifier and replace the string reference with symbol tale reference. With this approach, memory consumed by an identifier can be reduced from 12 bytes (Table II) to 8 bytes (Table V), and the interpreter task can be reduced as well. During the time of interpretation, it will be easier for the interpreter to take the decision whether to look up an identifier or not.

B. Dot Operator Node

The behavior of the dot operator is similar to that of the binary node. An object appears to the left of the Dot operator and the property of that object to the right of it (e.g. a.b). The number of binary nodes will increase if the dot operators are used in between two or more identifiers (e.g. a.b.c). Dot operators are used quite frequently. As it can be observed from Table II, a single dot operator occupies 24 bytes, where the property of that object is an identifier and the symbol table reference is NULL.

In this proposed solution, there are two node types instead of E_DOT_TOK and E_DOT_TOK_LIST. These are:

E_ATYPE_DOT_LEAF: If the dot operator is used between two identifier nodes then the left hand identifier node will have the symbol table reference if it is resolved, else if not resolved, it will have a string table reference and the right hand operator will hold only the string table reference.

E_ATYPE_DOT_NON_LEAF: For more than two identifiers, the left hand operator would have another AST node which is either E_ATYPE_DOT_LEAF node or E_ATYPE_DOT_NON_LEAF node and the right hand identifier has the string table reference only.

```
typedef struct _dot_leaft_node
{
        void *vASTNode; /* Identifier Node*/
        void *vStrTabRef ;
} DOT_LEAF_NODE ;
typedef struct _dot_non_leaft_node
{
        void *vASTNode; /* Dot Leaf/Dot Non Leaf
Node*/
        void *vStrTabRef ;
} DOT_NON_LEAF_NODE ;
```

C. Function Call Node

Functions are called with or without the arguments. Based on this behavior, function call node can be divided into two types.

E_ATYPE_FUNCTION_CALL_WITHOUT_ARGS: If the function call is without any argument then the left hand will be an AST node (E_ATYPE_DOT_LEAF or E_ATYPE_DOT_NON_LEAF or E_ATYPE_IDENTIFER_RESOLVED or E_ATYPE_IDENTIFER_NOTRESOLVED).

E_ATYPE_FUNCTION_CALL_WITH_ARGS: It is same as function call node, explained above. *(Table II)*

D. If-Then-else Call Node

E_ATYPE_IF_WITHOUT_ELSE: In the "if-then-else" statement, if there is no "else" statement, it will be represented as binary node instead of ternary node.

E_ATYPE_IF_WITH_ELSE: It is the same as if-then-else node, explained above. *(Table II)*

TABLE V
STRUCTURE SIZE CALCULATION

Operator	Memory Size		
	Structure Description	Size in Bytes	Total Size in Bytes
Identifier	E_ATYPE_IDENTIFER_RESOLVED/ E_ATYPE_IDENTIFER_NOTRESOLVED	4	8
	vNameRef/ vRefSymTab	4	
Function Call	E_ATYPE_FUNCTION_CALL_WITHOUT_ARGS	4	12
	Identifier Node	8	
Function Call	E_ATYPE_FUNCTION_CALL_WITH_ARGS	4	20
	Identifier (Function name)	8	
	Identifier (1 parameter)	8	
DOT Operator	E_ATYPE_DOT_LEAF	4	16
	pLeft -> vASTNode	8	
	pRight -> vNameRef	4	
DOT Operator	E_ATYPE_DOT_NON_LEAF	4	8+ (Depends on the number of identifiers)
	pLeft -> vASTNode	-	
	pRight -> vNameRef	4	

Operator	Memory Size		
	Structure Description	Size in Bytes	Total Size in Bytes
If-then-else	E_ATYPE_IF_WITHOUT_ELSE	4	12
	pC1-> conditional expression node	4	
	pC2–> then part statements	4	
If-then-else	E_IF_THEN_ELSE_TYPE	4	16
	pC1-> conditional expression node	4	
	pC2–> then part statements	4	
	pC3–>else part statements	4	

TABLE VI
STATISTICS OF MEMORY REPORT AFTER THE AST NODE OPTIMIZATION

Alexa Sites	After optimization		
	Memory without Optimization	Memory with Optimization	Total Memory Reduction (In Bytes)
CNN	630832	335760	295072
MSN	650756	343042	307714
EBAY	562724	305682	257042
REDIFF	25128	13016	12112
NDTV	225744	116652	109092
YATRA	513108	276718	236390
ALIBABA	651736	348200	303536
APPLE	900052	483122	416930
Go	257208	136760	120448

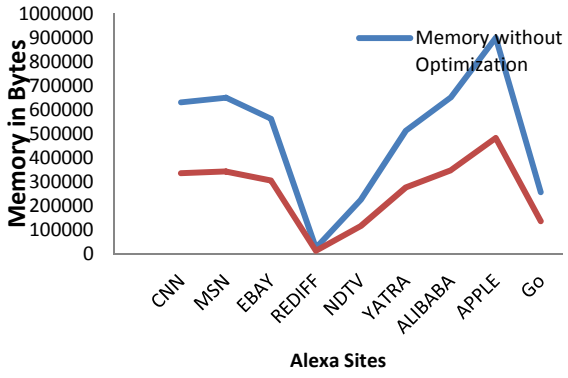

Figure 3. Statistics of Memory Report After AST Node Optimization

CONCLUSION

In this paper, we address the problem of optimization of the java script AST node in order to reduce the run-time memory requirements, with a set of with devices: a) Moto RAZR v3(brew 3.15), b) Qtopia (Linux OS), c) Samsung windows, and d) Nokia Series (Symbian OS) keeping in view that mobile devices are constrained with limited memory resources. The said optimization is carried with most frequently used in java script identifiers, function calls, dot operators and if-then-else statements. The experimental results justify the intended optimization [Table VI, Fig.3]. It can be useful for script engines.

ACKNOWLEDGMENT

We are extremely grateful to the administration of SOA Deemed to be University, Bhubaneswar, India for extending the opportunity to us in preparing this manuscript. We are at the same time thankful to Comviva Technologies, Bangalore, India, and in particular, Rajasekaran S, Kumar C, Vikaas BV, Zunder L, Arun T and the entire Sphinx team for providing the resources to carry out this research work successfully.

REFERENCES

[1] Prabhat Mishra, Aviral Shrivastava and Nikil Dutt, “Architecture Description Language (ADL)- driven software toolkit generation for architectural exploration of Programmable SOCs”, ACM Transaction on Design Automation of Electronic Systems”, Vol.11(3), pp.626-658, 2006.

[2] Aviral Shrivastava, Partha Biswas, Ashok Halambi, Nikil Dutt and Alexandru Nicolau, “Compilation Framework for code size reduction using reduced bit-width ISAs (rSAs)”, ACM Transaction on Design Automation of Electronic Systems, Vol.11(1), pp.123-146, 2006.

[3] Jongeun Lee and Aviral Shrivastava, “A Compiler – Microarchitecture Hybrid Approach to Soft Error Reduction for Register Files”, IEEE Transaction on Computer Aided Design, Vol.29(7), 2010

[4] Seungchul Jung, Aviral Shrivastava and Ke Bai, “Dynamic Code Mapping for Limited Local Memory Systems”, Proceedings of International Conference on Application-specific Systems Architectures and Processors (ASAP), pp.13-20, 2010.

[5] Jongeun Lee and Aviral Shrivastava, “A Compiler Optimization to reduce soft errors in register files”, Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES’09), pp.41-49, 2009.

[6] Amit Prabalkar, Aviral Shrivastava, Arun Kannar and Jongeun Lee, “SRDM: Simultaneous Determination of Regions and Function-to-Region Mapping for scratchpad mamories”, Proceedings of International Conference on High Performance Computing (HIPC 2008), pp.569-582, 2008.

[7] Aviral Shrivastava, Nikil Dutt, Alexandru Nicolau and Eugene Earlie, “Compiler-in-the-Loop ADL-driven Early Architectural Exploration”, TECHCON 2005: Semiconductor Research Corporation, 2005.

[8] Ashok Halambi, Aviral Shrivastava Nikil Dutt and Alexandru Nicolau, “A customizable Compiler Framework for Embedded Systems”, SCOPES, Springer, 2001.

[9] Easwaran raman and David I. August, “Optimization for Memory Hierarchies”, The Compiler Design Hand Book: Optimization and Machine Code Generation, Second edition, pp.5.1-5.28, 2008.

[10] Priti Shankar, “Instruction Selection using Tree Parsing”, The Compiler Design Hand Book: Optimization and

Machine Code Generation, Second edition, pp.17.1-17.35, 2008.

Sambit Kumar Patra is working as Senior Technical Lead in COMVIVA Technologies, Bengaluru from 2004. He completed his Master in Computer Application from NIELIT (formerly DOEACC Society) New Delhi in 2003. His research areas are Mobile and Telecom Technology, Compiler designing, Artificial Intelligence, Robotic and ad-hoc network. He is pursuing his postgraduate academic degree in Sikha "O" Anusandhan University, Bhubaneswar. His paper "Optimizing JavaScript Object Behaviour and Global Function Object", has been published in European Journal of Scientific Research (EJSR), Vol. 82, Issue. 3, pp. 397-406, July 2012.

Dr. Binod Kumar Pattanayak completed his M.S. in Computer Engineering from NTU Kharkov Poly Technical Institute, Ukraine in 1992. He was awarded Ph. D. in Computer Science and Engineering from Siksha 'O' Anusandhan University, Bhubaneswar, Odisha, India in 2011. He is currently working as Associate Professor and HOD in the Department of Computer Science and Engineering, Institute of Technical Education and Research, Siksha 'O' Anusandhan University, Bhubaneswar, Odisha, India. He has more than 25 research publications in reputed national and international journals and conferences. Five research scholars are pursuing their Ph. D. dissertation work under his supervision. Dr. Pattanayak is specialized in Computer Networks, Compilers, soft Computing and Computer Architecture.

Dr. Bhagabat Puthal started his career as an Asst. Professor in NIT, Rourkela **from** 1963 – 1981 for 18 years. Later he joined as Principal, Research Manager, Steel Authority of India Limited in 1981 and worked for 11 years. Then he worked as Professor, IGIT, Talcher **from** 1992 – 1997. Lastly Dr.Puthal joined ITER, SOA University, Bhubaneswar in the year 1997 and worked till 2011. Dr. Bhagabat Puthal was awarded PhD degree in, Microprocessors & Microcomputers from Sambalpur University, M.Tech from Indian Institute of Technology, Delhi (1979 – 1981) and B.Tech from Jadavpur University (1959 – 1963). He has to his credit a number of research publications in reputed national and international journals and conferences. A number of research scholars have pursued their Ph. D. research dissertation works under his guidance.