iJRASET
International Journal For Research in Applied Science and Engineering Technology

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 2 **Issue:** III **Month of publication:** March 2014

DOI:

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

ISSN: 2321-9653

INTERNATIONAL JOURNAL FOR RESEARCH IN APPLIED SCIENCE AND ENGINEERING TECHNOLOGY (IJRASET)

Compiler Optimization for SIMD type Vector Processor

Mohammad Suaib[1], Mohd. Akbar[2],

Department of Computer Science & Engg., Integral University Lucknow, India

Abstract— Performance of the processor can be enhanced by parallelization of instructions in terms of execution. Here we are applying compiler optimization techniques like loop unrolling, loop peeling for SIMD type Vector Processor. SIMD type vector processor is a high performance computational model which exploits the computational capabilities of both SIMD and vector architecture. SIMD type vector processor works on short vector instructions of vector length four and has four processing units which enables execution of four vector operands simultaneously [1]. To have the model which speeds up the computation in this paper we have created a CDFG (Control Data Flow graph) which gives the direct translation to the hardware. To create a CDFG we have used the tool SUIF2 and MACHSUIF which is a research project provided by Stanford University and Harvard university. To create a CDFG from the C source file first we have to do profiling on it to get the area of source file which have higher run time. We have unrolled the loops to get maximum ILP (Instruction level Parallelism) [7]. To limit the hardware we have unrolled only up to a maximum size of 4 [1]. Then a CFG (Control Flow Graph) is created for the source C file. Then the data flow analysis has been done on each basic block of the CFG by using the bvd class provided with MachSUIF. Then all the DFG (Data Flow graph) of every basic block are combined to get a full CDFG. In this paper We have designed the new hardware i.e. the SIMD type Vector processor based on HPL-PD VLIW architecture ISA which is supported by trimaran by default to accelerate the work.

Keywords— SIMD-Vector Architecture, Loop unrolling, ILP, Suif, Machsuif, Loop peeling

I. INTRODUCTION

A compiler is a computer program that transforms source code written in a programming language i.e. the source language into another computer language i.e. the target language, often having a binary form known as object code [2,3]. If the compiled program can run on a computer whose CPU or operating system is different from the one on which the compiler runs, the compiler is known as a cross-compiler. A program that translates from a low level language to a higher level one is a decompiler. A program that translates between high-level languages is called a language translator, source to source translator, or language converter. A language rewriter is a program that translates the form of expressions without a change of language .document is a template.

The output of some compilers may target hardware at a very low level, for example a Field Programmable Gate Array (FPGA) or structured Application-specific integrated circuit (ASIC) [8]. Such compilers are known as hardware compilers or synthesis tools because the source code they compile effectively control the final configuration of the hardware and how it operates; the output of the compilation are not instructions that are executed in sequence - only an interconnection of transistors or lookup tables. For example, XST is the Xilinx Synthesis Tool used for configuring FPGAs [8]. Similar tools are available from Altera, Synplicity, Synopsys and other vendors.

A. *Compiler optimizations*

Compiler optimization is the process of tuning the output of a compiler to minimize or maximize some attributes of an executable computer program [4]. The most common requirement is to minimize the time taken to execute a program; a less common one is to minimize the amount of memory occupied. The growth of portable computers has created a market for minimizing the power consumed by a program. Compiler optimization is generally implemented using a sequence of optimizing transformations, algorithms which take a program and transform it to produce a semantically equivalent output program that uses fewer

ISSN: 2321-9653

resources. Some code optimization problems are NP-complete, or even undecidable. In practice, factors such as the programmer's willingness to wait for the compiler to complete its task place upper limits on the optimizations that a compiler implementer might provide. Optimization is generally a very CPU- and memory-intensive process.

B. *Instruction-level parallelism (ILP)*

Instruction-level parallelism (ILP) is a measure of how many of the operations in a computer program can be performed simultaneously [5,6]. A goal of compiler and processor designers is to identify and take advantage of as much ILP as possible. Ordinary programs are typically written under a sequential execution model where instructions execute one after the other and in the order specified by the programmer. ILP allows the compiler and the processor to overlap the execution of multiple instructions or even to change the order in which instructions are executed [7].

C. *Loop Unrolling*

Loop unrolling is effective in uncovering ILP by exposing more parallelism to the compiler or underlying hardware [5]. It is, in essence, a technique that transforms loop level parallelism into instruction level parallelism.

```
for (i=0; i<8; i++) {

a[i] = b[i] + c[i];

}
```

Before loop unrolling.

```
for (i=0; i<8; i+=4) {

a[i+0] = b[i+0] + c[i+0];

a[i+1] = b[i+1] + c[i+1];

a[i+2] = b[i+2] + c[i+2];

a[i+3] = b[i+3] + c[i+3];

}
```

After loop unrolling.

D. *CFG (Control Flow graph)*

A control flow graph is a representation of a program where contiguous regions of code without branches, known as basic blocks, are represented as nodes in a graph and edges between nodes indicate the possible flow of the program [9,4]. In a control flow graph each node in the graph represents a basic block, i.e. a straight-line piece of code without any jumps or jump targets; jump targets start a block, and jumps end a block. Directed edges are used to represent jumps in the control flow.

E. *DFG (data Flow graph)*

The purpose of data-flow analysis is to provide global information about how a procedure or a larger segment of a program manipulates its data [4,10]. Constant-propagation analysis seeks to determine whether all assignments to a particular variable that may provide the value of that variable at some particular point necessarily give it the same constant value.

II. PROPOSED HARDWARE MODELING FOR COMPILATION

This section describes the proposed model for creating a CDFG from the basic C based applications. Here the basic tools are SUIF2 and the MACHSUIF is also described in detail to help in understanding of the creation of CDFG. The CFG pass ant the CDFG pass used in the implementation is described along with its psuedocode which is written in MACHSUIF.

A. *SUIF Overview*

The SUIF2 compiler system can be described as two major parts: the front-end and the back-end as in Fig.1 The operations of the front-end consist of lexical analysis, syntax analysis, semantic analysis, and generation of the SUIF intermediate format file, file type is filename.suif. In the back-end, the operations are code optimization and code generation. The SUIF2 compiler system gives full supports of handling the SUIF intermediate format file. By applying the SUIF intermediate format file and related libraries provided by the SUIF2 compiler system, the analysis and transformation of a

ISSN: 2321-9653

sequential program can be made easily. The SUIF2 compiler system is built of many modules and a common driver called suifdriver. Fig.2 shows the SUIF2 compiler pass which is formed with importing modules dynamically via the suifdriver. The SUIF2 compiler system allows user to develop new modules or compiler passes according to their specific requirements. Furthermore, programming languages that SUIF2 compiler system supports in current are as follows, Fortran, C, C++ and Java.

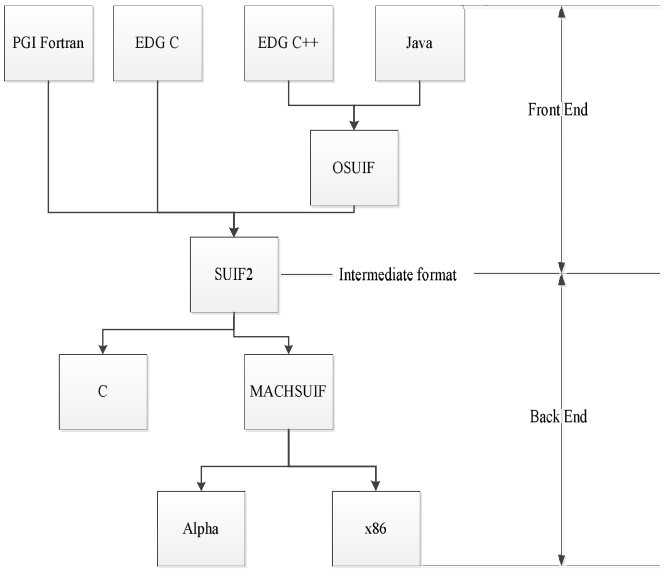


Fig.1 SUIF2 compiler system

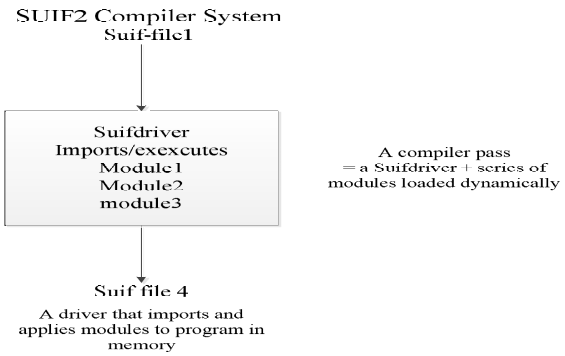


Fig. 2 SUIF2 compiler pass

B. *MachSUIF*

Machine SUIF is a flexible, extensible, and easily-understood infrastructure for constructing compiler back ends. Machine SUIF is both a working compiler and a collection of analyses/optimizations that can be quickly and easily inserted into a new compilation environment. Machine-SUIF distribution contains a working compiler based on the Stanford SUIF compiler infrastructure (version 2.1). This compiler is capable of producing optimized code for machines based on the Alpha or x86 architectures.

C. *CDFG Pass*

To have the data flow graph the class bvd of the MachSUIF is used. The bit-vector data flow (BVD) library of Machine SUIF is a framework for iterative, bit-vector-based data-flow analyzers. It uses Machine SUIF's control-flow graph (CFG) library to parse the program being analyzed into basic blocks, and it associates data-flow results with the CFG nodes.

The BVD library contains two concrete solvers, one that computes liveness information and another that does reaching-definitions analysis. The data-flow analyzers do two things: they identify an interesting class of syntactic or semantic program elements, and for each such element, they identify the points in the program to which the element, or some aspect of the element i.e. flows. For an available-expressions problem, the elements are syntactic expressions evaluated by the program, and an expression e flows to point p in the program if e is computed (generated) on every path to p without being invalidated (killed) by an assignment to some variable in e before p is reached. For the reaching-definitions problem, the universe of interesting elements consists of the statements that assign to, or otherwise side-affect storage locations, and these definitions flow to all the program points reached by paths that don't contain an overriding assignment. For a liveness problem, the universe consists of storage cells, such as variables and registers. The liveness of a variable is generated by a use of the variable, and it flows backward along control paths until killed by a definition of i.e. an assignment the variable.

D. *CDFG Generation*

In this paper we are concentrating to extract ILP from C based applications which can be computed in a SIMD machine which exploits the ILP. Loop unrolling and loop peeling is done to find whether the code is vectorizable or to improve performance. So here a loop is taken and it is unrolled and peeled according to the proposed architecture to bring out its

ISSN: 2321-9653

effectiveness. The loop unrolling and peeling is done by SUIF2 passes. To implement the proposed work we have used MachSUIF. In MachSUIF, the IR(Intermediate Representation) uses a SUIFvm (SUIF virtual machine architecture) which assumes that the underlying architecture is a generic RISC which is not biased towards any existing architecture.

The program code is decomposed into its IR consisting of operations with minimal complexity i.e. in primitive or atomic instructions. Then this IR description of the program code is organized into control data flow graph with primitive instructions as its nodes and edges denoting control and data dependencies [11]. Before the CDFG is created loop unrolling and peeling is done. The flow graph of the CDFG generation is shown in fig.1. The shaded blocks on the figure shows the available passes of the MachSUIF infrastructure. The c2suif pass converts the input ANSI C code to the SUIF frontend i.e. the code is preprocessed and its SUIF representation is emitted. After c2suif loop unrolling and peeling is done by a custom made pass which does it according to our architecture. In this pass at first the SUIF file is converted to the ANSI C code and then the loop unrolling and peeling is done on this C code. The peeling is done in such a manner that the number of iterations would become a multiple of 4 and unrolling is done for a size of 4. After the unrolling and peeling the pass again converts the C code to SUIF IR. The 3rd step is do_lower pass or an equivalent pass with all necessary transformations which is provided with SUIF. In this step several machine independent transformations is done like dismantling of loop and conditional statements to low-level operations. By doing the do_lower we translate the higher SUIF representation to lower SUIF representation. To convert lower SUIF representation to SUIFvm representation an s2m compiler pass is used which is available in MachSUIF. After s2m, architecture independent optimizations are done on IR and a CFG (Control Flow Graph) is created by pass il2cfg. A CFG form is got in which the cdfg2dot pass parses on each node of the CFG and constructs a corresponding CDFG. By this we get a CDFG of each node.

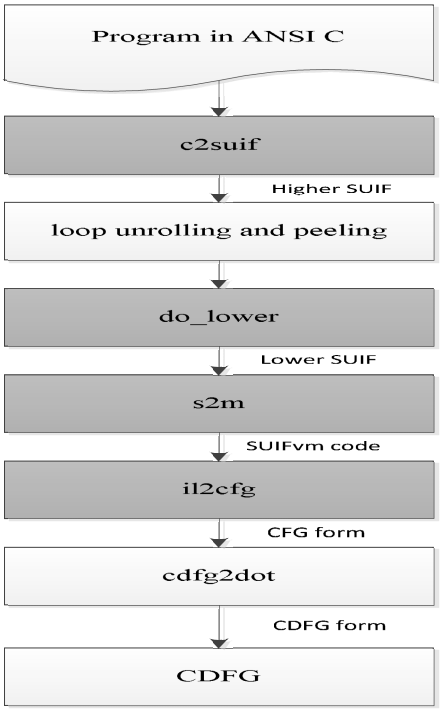


Fig. 3 Flow graph for CDFG generation

IV. COMOILER FLOW FOR THE SIMD VECTOR PROCESSOR ARCHITECTURE

To execute software code on the new architecture i.e. a VLIW processor with a tightly coupled coprocessor which is a SIMD type Vector model, a compilation flow was developed. The input to the flow is C code. The code is read into the compiler and scheduled into instructions that can be performed in parallel. The final result which is a elcor IR which can be executed on the new hardware and is simulated in the simulator SIMU provided with the Trimaran to collect the binaries or executables of the C based application.

Trimaran was selected as a framework for the SIMD VLIW compiler. Trimaran was chosen because it is an open source, extensible VLIW compiler. It contains separate frontend and back-end code and code that translates the intermediate representation, IR, between the two. Third-party back-ends exist to target other architectures. Triceps [12] is a Trimaran back-end to target the StrongARM processor, and Tritanium [13] is available to target the Intel Itanium series of processors. Trimaran was also chosen because the projects mentioned above serve as examples of how to modify the code generation for different processor targets.

Trimaran has a back-end vectorizer that can identify and exploit data level parallelism for efficient execution on architectures with SIMD (single-instruction multiple-data) support. The technique implemented in Trimaran is called selective vectorization [14]. It creates highly efficient instruction schedules by distributing computation between scalar and vector functional units to improve resource utilization. For processor models that contain an abundance of scalar and vector processing units, selective vectorization creates loops with a balance of both vector and scalar instructions. Since vector and scalar occupy the same loop body, scalar operations are unrolled by a factor of the vector length. As such, the technique is most applicable to the short-vector instruction sets commonly found in today's multimedia extensions. The Elcor parameter do_vectorize in $ELCOR HOME/DRIVER DEFAULTS enables the vectorization of loops during compilation. The vectorizer is most effective, and in many cases only applicable, when it has precise memory dependence information. 4 options are found in $ELCOR HOME/VECTORIZER DEFAULTS to provide control in applying the vectorizer:Presently the Trimaran supports 4 vectorize model. The vectorizer will apply different techniques and heuristics according to the model selected. There are four current models implemented. They are:

– Unroll the loop by a factor of vector length.

– Vectorize all vectorizable operations.

– Perform full vectorization or no vectorization. Select the option with the highest predicted performance after modulo scheduling.

– Perform selective vectorization.

The SIMD Vector model back-end is closely coupled to code from the ELCOR back-end. This is done by having a new .hmdes2 file which specifies our architecture also. The architecture to be defined is shown in figure 5.3. In this .hmdes2 file we have added the functionality vectorization also by enabling the vectorization in the Trimaran. The proposed architecture is a SIMD VLIW architecture which is a SIMD Vector model as a tightly coupled co-processor of a VLIW machine. The VLIW is chosen in the work as it is already present with Trimaran and thus speeds up the architecture modelling process.

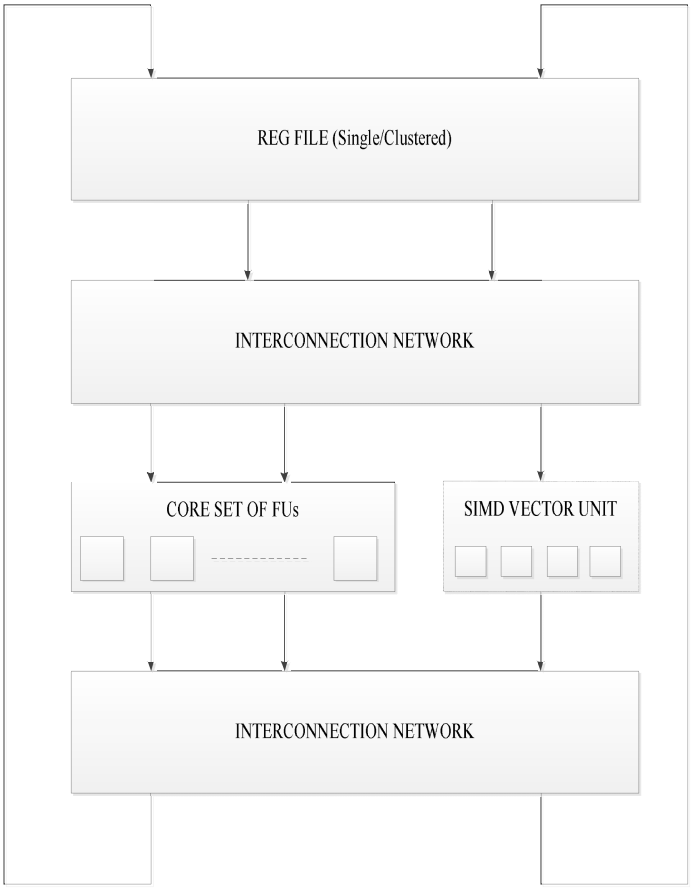


Fig. 4 The enhanced architecture.

We have made the number of vector units as 4 as proposed architecture supports a 4- way short vector instructions. Vec length specifies the vector length in number of elements. In this thesis as 4-way short vectors are taken the vector length is fixed as 4. Currently, all vector registers must have the same length (heterogeneous vector lengths are not supported). vir static size and vir rotating size: This specifies the number of static and rotating vector integer registers. Each register actually consists of vec length integer elements. vfr static size and vfr rotating size: This specifies the number of static and rotating vector floating-point registers. Each register actually consists of vec length floating-point elements. vec integer units, vec integer perm units, vec integer xfr units: These specify the number of functional units available for vector computation, vector permutation operations, and vector-scalar transfer operations. Here all the functional units like integer units, float units, permutation units, and subroutine units are kept as 4 as the hardware model proposed is a SIMD type

vector processor which is able to operate on 4 instructions at a time. Only changes that were required for the SIMD Vector machine were made. The back-end is responsible for instruction scheduling, register allocation, and code generation.Further the vector instructions are defined along with its latencies and other properties in the machine description provided with Elcor in Trimaran. By running the elcor we will be provided with an elcor IR which is run in simu to produce the binaries or executables.

V. CONCLUSION

In this paper we have shown how the binaries are generated from the benchmark or the C based applications using the tool Trimaran. The elcor backend was modified to incorporate the custom hardware which has been generated from the CDFG which was described in the previous section. Here the custom hardware is used as a tightly coupled co-processor as a SIMD type vector processor with the basic hardware which is a single cluster VLIW machine.

VI. REFERENCES

[1] Mohammad Suaib, Abel Palaty, and Kumar Sambhav Pandey,"Architecture of SIMD Type Vector Processor", Digital Library URI: http://www.ijcaonline.org/archives/- volume20/number-4 /2418-3233, ISBN: 978-93-80749-15-7, 2011.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, Compilers Principles, Techniques, and Tools, Addison-Wesley, 1986.

[3] Randy Allen and Ken Kennedy, Optimizing Compilers for Modern Architectures: A Dependence-based Approach, Morgan Kaufmann, San Francisco, California, 2001.

[4] Steven S. Muchnick, ADVANCED COMPILER DESIGN AND IMPLEMENTATION, Morgan Kaufmann, 1997.

[5] Davidson, J. W. and Jinturkar S.,"Improving Instruction level Parallelism by Loop Unrolling and Dynamic memory Disambiguation," Technical Report CS-95-13, Department of Computer Science, University of Virginia, Charlottesville, February 1995.

[6] Bacon, D. F., Graham, S. L., and Sharp O. J.,"Compiler Transformations for High-Performance Computing," ACM Computing Surveys, 26(4), Dec. 1994, pp. 345-420.

[7] Abel Palaty, Mohammad Suaib, and Kumar Sambhav Pandey ,"Exploiting ILP in a SIMD type Vector Processor," ACC-2011, will Communications in Computer and Information Science Series(CCIS), ISSN: 1865:0929.

[8] Takahisa Wada, Shunichi Ishiwata, Katsuyuki Kimura, Keiri Nakanishi, Masato Sumiyoshi,Takashi Miyamori, Member, IEEE, "A VLIW Vector Media Coprocessor With Cascaded SIMD ALUs" IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS, VOL. 17, NO. 9, SEPTEMBER 2009

[9] Robert Morgan, Building an Optimizing Compiler. Digital Press, 1998.

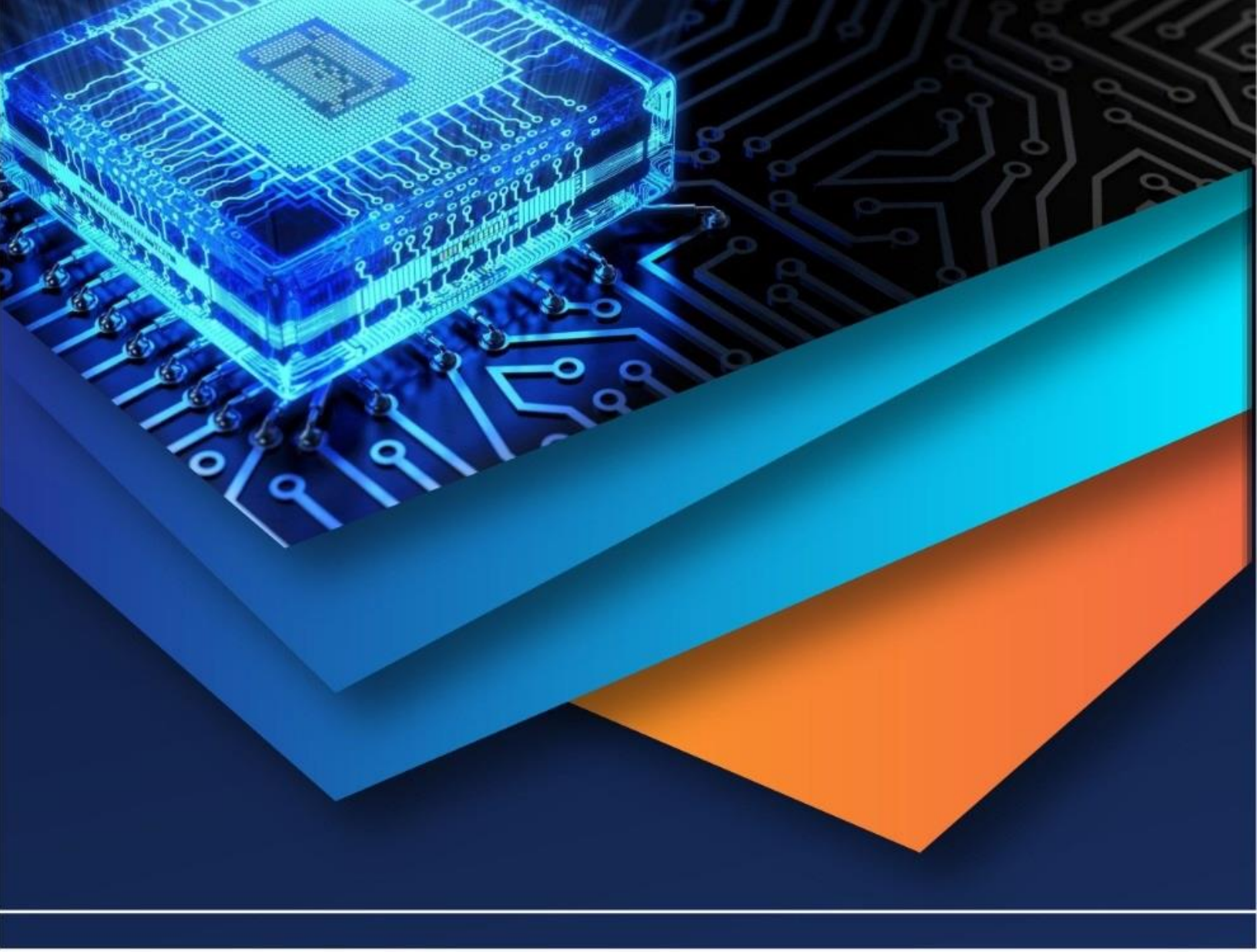
[10] Glenn H. Holloway and Michael D. Smith, The Machine-SUIF Control Flow Graph Library, The Machine SUIF documentation set, Harvard University, 1998.

[11] Nikolaos Kavvadias and Spiridon Nikolaidis,"Application Analysis with Integrated Identification of Complex Instructions for Configurable Processors,"Proc. of the 14th Intl. Workshop on Power and Timing Modeling, Optimization and Simulation, Santorini, Greece, pp. 633-642, September 15-17(2004).

[12] L. N. Chakrapani, W. F. Wong, and K. V. Palem,"Enhancing the Trimaran compiler infrastructure to support strongarm code generation,"CREST, Georgia Institute of Technology, Atlanta, GA, Tech. Rep. CREST-TR-01-01, 2001.

[13] Y. Chobe, B. Narahari, R. Simha, and W.Wong,"Tritanium: Augmenting the Trimaran compiler infrastructure to support ia64 code generation,"The George Washington University, Washington DC.

[14] X. Tang, T. Jiang, A. Jones, and P. Banerjee,"Compiler optimizations in the pact hdl behavioral synthesis tool for asics and fpgas," IEEE International SoC Conference (IEEE-SOC), September 2003.



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY