Vectorization, Obfuscation and P4 LLVM Tool-chain

Dangeti Tharun Kumar

A Thesis Submitted to

Indian Institute of Technology Hyderabad

In Partial Fulfillment of the Requirements for

The Degree of Master of Technology

भारतीय प्रौद्योगिकी संस्थान हैदराबाद
Indian Institute of Technology Hyderabad

Department of Computer Science and Engineering

July 2018

Declaration

I declare that this written submission represents my ideas in my own words, and where ideas or words of others have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be a cause for disciplinary action by the Institute and can also evoke penal action from the sources that have thus not been properly cited, or from whom proper permission has not been taken when needed.
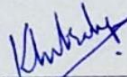
_____
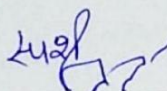(Signature)

_____
(Dangeti Tharun Kumar)

CS15MTECH11002

(Roll No.)

Approval Sheet

This Thesis entitled Vectorization, Obfuscation and P4 LLVM Tool-chain by Dangeti Tharun Kumar is approved for the degree of Master of Technology from IIT Hyderabad
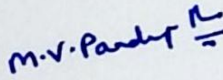
(PROF. SUBRAHMANYAM KALYANASUNDARAM_____) Examiner
Dept. of Computer Science and Engineering
IITH

(PROF. SPARSH MITTAL_____) Examiner
Dept. of Computer Science and Engineering
IITH

(Dr. Ramakrishna Upadrasta) Adviser
Dept. of Computer Science and Engineering
IITH

(PROF. M.V.P. RAO_____) Chairman
Dept. of Computer Science and Engineering
IITH

Acknowledgements

I would first like to thank my thesis advisor Dr. Ramakrishna Upadrasta, for his continues encouragement and support throughout my stay at IITH. I know no words to express my gratitude towards him. I would like to thank my co-authors Utpal, Santanu, Anirudh, and Venkata. Their collaboration has lead to many of the ideas and results presented here.

Also, I want to extend my thanks to the Compilers group at IITH. I would also like to acknowledge G. Srikanth Kumar, my friend at WSU as the second reader of this thesis. I am gratefully indebted to him for his very valuable comments on this thesis. I must express my very profound gratitude to my parents for providing me with unfailing support throughout my years of education.

Abstract

This thesis broadly focuses on three different areas: Loop Vectorization, Code Obfuscation, and P4LLVM compiler. The work in Loop vectorization starts with a comparison of Auto-vectorization of GCC, ICC and LLVM compilers and show their strengths and weakness. As an attempt to improve LLVM's Auto-vectorization, we propose to improve Loop Distribution using exact dependences from Polly. Our work on Loop Distribution shows promising results. We developed an LLVM based Code Obfuscation engine with various obfuscation techniques as transformation passes, our techniques are novel and are different from existing works [1]. In hardware circuit obfuscation several methods were proposed at the hardware level to secure the IP. Our approach is to obfuscate the circuits at the software level, using code obfuscation techniques.

Contents

Chapter 1

Study of Auto-Vectorization in Compilers

1.1 Introduction

Automatic vectorization is an important phase of compiler optimization. It involves automatically detecting sections of the input program by the compiler, that can be translated to vector instructions. In this work, we make a study of the current vectorization features which has been implemented in the present day compilers. We present a detailed analysis and comparison of three compilers: LLVM (Low Level Virtual Machine) Compiler Infrastructure, GCC (GNU Compiler) and ICC (Intel Compiler), highlighting their strengths and their weaknesses. We have used TSVC [2] benchmark exclusively designed for testing vectorization capabilities of compilers.

In the era of High Performance Computing [3], various methodologies have been explored to run programs in the most efficient way in the emerging architectures. Execution of programs in a parallel fashion has almost become a necessity and hence is an important research area now-a-days. To enable parallelism, the modern processors come with various varieties of hardware features: multi/many-cores, GPUs, special registers, and even specialized hardware elements are being added to current generation machines. Various compilers have been targeting each of these new hardware features, and SIMD(Single Instruction Multiple Data) class of parallelization is a good example where ISA is extended to support vector arithmetic. Single Instruction Multiple Data (SIMD) deals with processing of multiple data elements using single instruction and modern processors have support for up to 512 bit vector operations. Our work focuses on a type of SIMD vectorization [4] the problem of automatic vec-

torization of programs within the compiler framework which makes use of supporting hardware architectures like Streaming SIMD Extensions (SSE).

1.2 Study of Vectorization

In our experiments with the TSVC benchmark suite, out of 151 loops, the number of loops vectorized by LLVM, GCC, and ICC are 70, 82, and 112 respectively. In Fig. 1.1 we computed the relative performance of the three compilers with respect to the TSVC benchmark suite. There are 31 loops which are not vectorized by any of
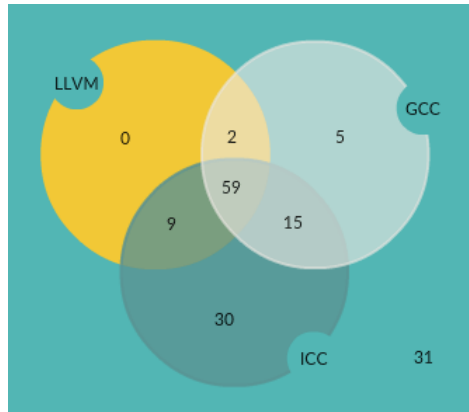

Figure 1.1: TSVC results

the compilers. But, 59 loops are vectorized by all of them. The Intel compiler (ICC) is able to transform significantly more number of loops exposing vectorization and perform partial vectorization for some of these loops. We find that ICC alone is able to vectorize 30 loops on which the other two compilers fail. Also, LLVM performs quite poorly in this test, as it is unable to vectorize many loops which are successfully vectorized by other two compilers. The reason may be that LLVM is a relatively new compiler and all the features are not integrated yet. The superior performance of ICC *may be* due to the fact that the same organization that writes the compilers also makes the hardware on which it was tested. So, it is possible that their compiler and hardware are more in sync with each other to achieve maximum performance. The performance of GCC relatively average as it fails to detect many optimization opportunities as compared to ICC. We also recorded the runtime of each loop in TSVC by running each of them ten times and calculating their mean values.

The 31 loops which were not vectorized by any of the compilers have loops containing complex control flow with if-else ladder (switch statements are also not considered), complicated access patterns of array locations (non-uniform accesses, indirect
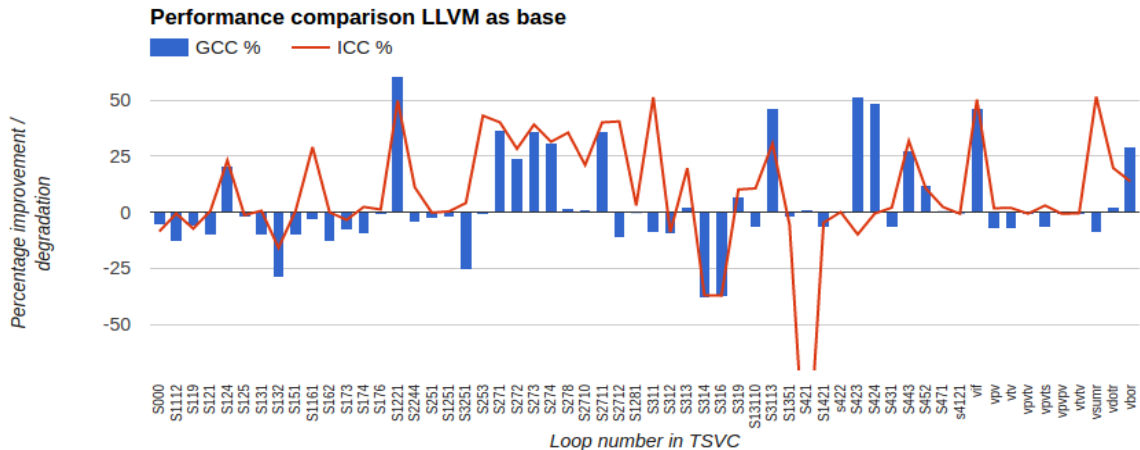
Figure 1.2: A comparison of run times of only those loops vectorized by all three compilers with LLVM as base line. X-axis is the loopId as given in TSVC and Y-axis is percentage improvement/degradation over LLVM.

accesses), loops with stride length more than one, failure to do scalar expansion, statements with recurrence relations and loops containing goto instructions. A reason for compilers not able to vectorize the above said loops is the absence of sophisticated hardware to load/store data from non-contiguous locations effectively. In addition, greater strides between data in memory renders vectorization inefficient. For this case, the cost model of the compiler derives that vectorization will not improve performance. Loops with multiple exits are rendered non-vectorizable as trip count cannot be determined. Switch statements can be represented as an if-else ladder but cannot be flattened. They are not supported for vectorization in any of the three compilers, which is a good area for future research.

We also present a comparison of the runtime of the loops successfully vectorized by all the compilers in the Fig. 1.2. we show the relative runtimes of GCC and ICC with respect to LLVM. The base line is LLVM and the blue bar is GCC and red curve is ICC. We find that for some cases, the vectorized run times are significantly different among the three compilers.

1.3 Conclusion

Our work compares the three major compilers for Auto-Vectorization capabilities. We tried to present various possible reasons for Loops not getting vectorized, thereby exposing the areas that needs improvement. To the best of our knowledge, this is the first work in comparing vectorization in compilers.

Chapter 2

Loop Distribution for Vectorization in LLVM

2.1 Introduction

We propose to improve Loop Distribution in LLVM targeting improvement of inner-loop vectorization with the use of precise Polyhedral Dependence analysis from Polly (Polyhedral engine in LLVM). Our approach is to construct an LLVM IR level dependence graph from Polly's dependences and use them for Loop Distribution. We use classic Maximal Loop Distribution algorithm. Our loop distribution pass shows promising results on the TSVC [2] benchmark; it is able to distribute 11 loops, while the earlier distribution pass in LLVM is unable to distribute at all. We also have performance numbers from SPEC CPU 2017 C and C++ benchmarks with LLVMs dependence analysis *Loop Access Info* and with the new instruction level DA .We believe that our work is the first step towards *scalable* and *pre-defined* loop-transformations in LLVM using exact dependences from Polly.

In this work, our goal is to improve loop vectorization in LLVM using *Polly*, the polyhedral optimizer in LLVM. Though the analysis and transformations of Polly are based on a strong mathematical framework, they come with additional compile-time. It is well understood that the major component of compile time of Polly is the scheduling algorithm of Polly that relies on solving a large LP/ILP problem that uses underlying Integer Set Library(ISL) library in Polly. In our current work, we attempt to alleviate this problem by using the exact dependences of Polly for well-defined individual transformations without the compile time cost that comes with it.

```
for(int i = 4; i < 1000; i++) {
  a[i] = a[i - 1] * 2;
  b[i] = c[i - 4] * c[i];
}
```

Figure 2.1: code snippet with 2 RAW dependences

```
RAW dependences:
{
    Stmt_for_body[i0] -> Stmt_for_body[4 + i0] : 0 <= i0 <= 991;
    Stmt_for_body[i0] -> Stmt_for_body[1 + i0] : 0 <= i0 <= 994;
}
```

Figure 2.2: Dependences represented as ISL sets

Examples of such well-defined transformations are loop-distribution, statement/instruction reordering, modulo-scheduling, etc. We obtain the dependence information from Polly using our interface and integrate with the intra-iteration and data-flow dependences from the LLVM infrastructure to construct the dependence graph. We use this dependence graph to do the transformation.

2.2 Construction of RDG from Polly's Dependences

We use *PolyhedralInfo* [5] an analysis pass in Polly which exposes interfaces for checking parallel and vectorizability of loops. Polly represents dependences in the form of Integer Set Library(ISL) sets which cannot be used directly for analysis in LLVM. For example Figure 2.1 is a code snippet with 2 RAW dependences, Figure 2.2 depicts the ISL set representation of dependences. We have extended *PolyhedralInfo* to expose dependence information as a Reduced Dependence Graph (RDG) $G(V, E)$ [6], whose vertices $v \in V$ represents a statement S_i in the high level language and each edge $e \in E$ represents a dependence from S_i to S_j. The edge between nodes is annotated with the following information

- Dependence Distance: The iteration gap between dependent instructions in a loop.

- Dependence Level: At what depth of the loop the dependence exists.

- Type: RAW, WAR and WAW

2.3 Classic Loop Distribution Algorithm

Loop distribution is the technique of dividing a single loop into multiple loops, each of which iterates over a distinct subset of statements in the original loop body. This transformation improves parallelism, both SIMD and ILP, as well as improving cache locality. The below is the generalized loop distribution algorithm using reduced dependence graph (RDG) proposed by Allen and Kennedy [6].:

1. Compute the Strongly Connected Components (SCC) over the RDG.

2. Perform a topological ordering of the SCCs.

3. Relocate all the statements in every SCC to a new loop.

Hence, all the statements S_i involved in a dependence cycle are part of a single distributed loop, and all other statements are part of their own individual loops, thereby exposing parallelism.

2.4 Few details of Implementation in LLVM

A statement is S_i is defined as a map containing of $I_i \rightarrow \{\text{set of } I_k\}$, where I_i is generally a store instruction and I_k is an instruction with data flow dependence to the store instruction I_i. Statements without a store instruction (orphans) needs special handling.

The vertex of the graph is chosen to be a statement rather an LLVM IR instruction, because of the following reasons:

1. Polly considers a basic block to be a single statement as of now, the dependences inside a basic block are not recognised.

2. The dependence information from Polly does not contain data dependences i.e, use-defs. We tried to pack them together in a statement.

3. Statement representation helps in movement of the code to a new loop while transformation.

We make the loop instructions into statements and then obtain the dependences from PolyhedralInfo and construct the graph. Then we add the intra iteration dependences for which we depend on LLVM's LAI(Loop Access Info). Now that the dependence graph is constructed, we find the SCCs and transform the code.

2.5 Results

We evaluated our implementation on TSVC [2] and SPEC-CPU 2017 benchmarks. The improved loop distribution pass could successfully distribute 11 new kernels in TSVC, while the existing pass (`-enable-loop-distribute`) does not distribute any.

Figure 2.3 and 2.4 shows the performance improvement of Loop Distribution with Polly's and LLVM's dependence analysis on TSVC benchmark.

Figure 2.5 and 2.6 shows the performance improvement of Loop Distribution with Polly's and LLVM's dependence analysis on SPEC SPEED 2017.
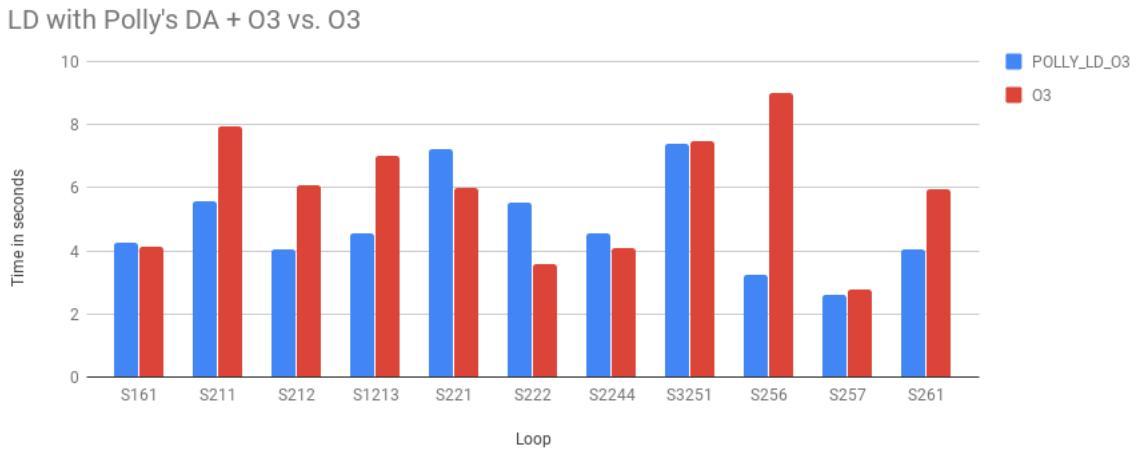

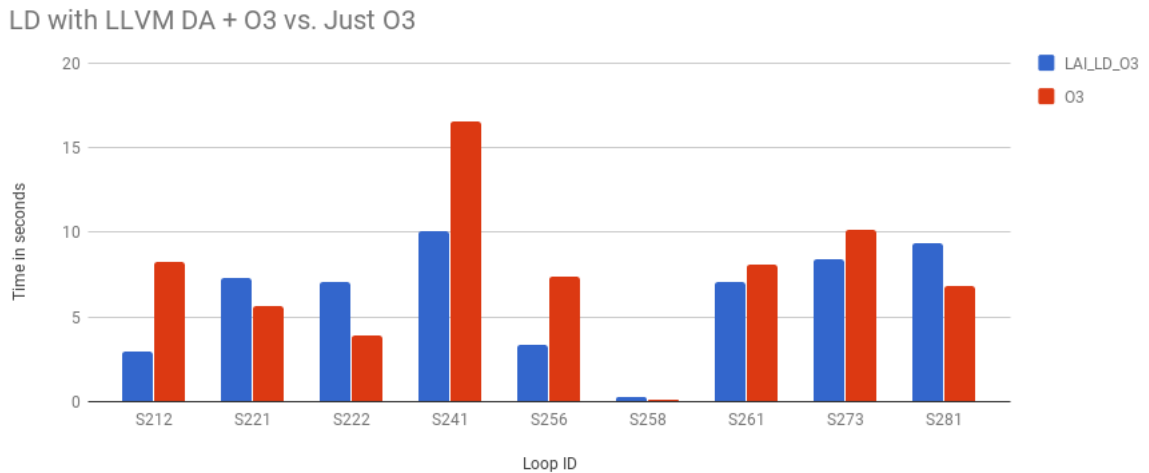Figure 2.3: Loop Distribution with Polly's DA + O3 vs. Just O3 on TSVC

Figure 2.4: Loop Distribution with LLVM LAI DA + O3 vs. Just O3 on TSVC
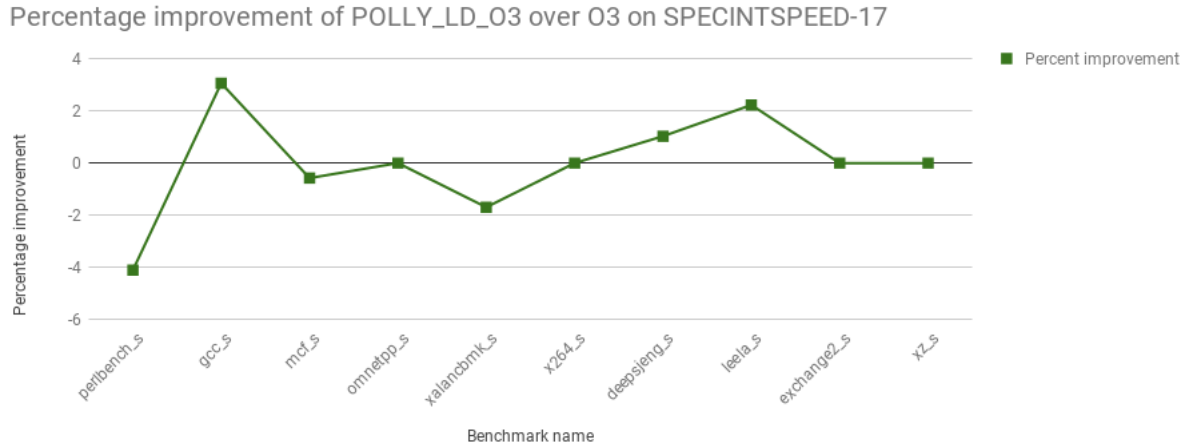
Figure 2.5: Loop Distribution with Polly's DA + O3 vs. Just O3 on SPEC INT SPEED
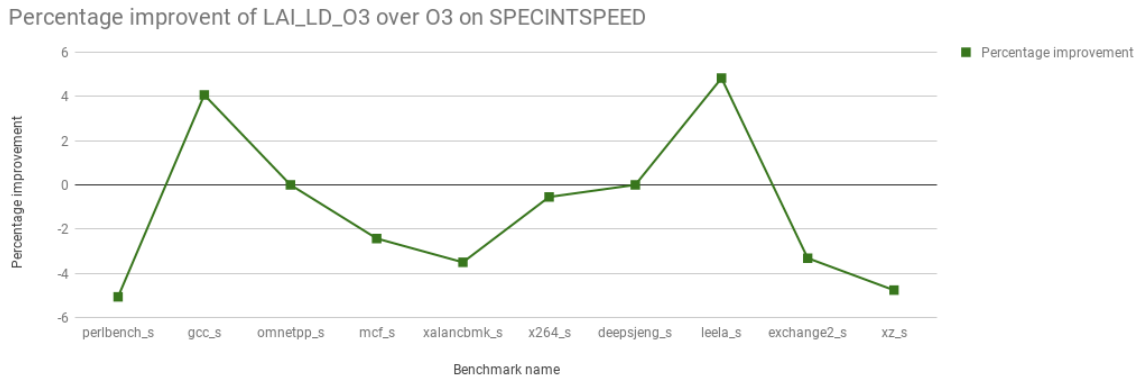
Figure 2.6: Loop Distribution with LLVM LAI DA + O3 vs. Just O3 on SPEC INT SPEED

2.6 Conclusion

Our current distribution heuristic is simplistic and aggressive (we distribute whenever it is legal) and this results in degradation of performance in loops from both TSVC and SPEC. A cost-model is essential for the algorithm to perform distribution only when it is beneficial. Future work involves designing a cost model and incorporating control dependences so that loops with these kind of dependences can also be distributed.

Chapter 3

Code Obfuscation

3.1 Introduction

Security is of vital importance in commercial and defense software development. Many commercial products are prone to adversarial attacks in the form of logic sniffing, license break etc. Reverse engineering is a process of reproducing another manufacturer's product by a detailed examination of its construction. Code obfuscation is the technique of transforming the code in a way that hardens the job of a reverse engineer. The purpose of obfuscation is to increase the time for understanding the code behavior without sacrificing much performance. In this chapter we discuss about various techniques we have implemented in LLVM.

3.2 Related work

OLLVM is a similar work from Junod et al. [1], they have implemented techniques like instruction substitution, Bogus control flow, Control flow flattening etc. Only a part of their work is open-source. The *Tigress* is another tool that performs obfuscation on C language unlike a more generic work like OLLVM.

3.3 Function Argument Folding

The goal of this pass is to modify the parameters of user defined functions into char* argv[] format. We modify the function signature and the call/invoke instructions to these functions.

```
1
   int fun(char *arg[]) {
3      int a = arg[0];
       int b = arg[1];
5      char c = (char)arg[2];
       // do something
7 }
```

```
1
  int fun(int a, int b, char c) {
3     // do something
  }
```

(a) Original code   (b) Obfuscated code
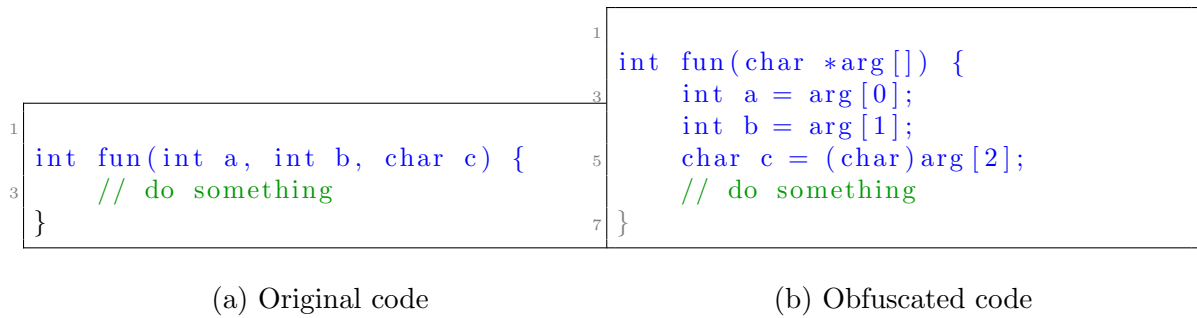
Figure 3.1: Function Argument Folding

3.3.1 Cases where arguments are not folded

In the following cases function arguments are not folded:

- Function is the main function in IR.

- Function has only declaration, no definition in the module.

- Function with variable number of arguments.

- Function has at least 1 argument.

- A function pointer points to the function.

3.4 Loop Index Indirection

In this technique we substitue integer induction variable of a loop with array and index combination. This leads to an indirect access to the induction variable. The code snippet in Figure 3.2 indirect access.

10

```
1  for(int i=1; i<100; i+=3)          1  for(int j=0; j<33; j++)
3  {                                   3  {
       cout<< i <<endl;                       cout << arr[j] <<endl;
5  }                                   5  }
```

(a) Original code   (b) Obfuscated code

Figure 3.2: Loop Index Indirection

3.4.1 Loop selection criteria

The following conditions should be satisfied for a loop to be considered for index indirection:

- Should be an innermost loop.

- Has statically computable trip count by Scalar Evolution. This is required for allocating array for indirect access.

- There should be at least 1 integer induction variable in the range *i[1-64]*. Increase in induction variables leads to creation of more arrays which is a memory overhead.

3.5 Loop Reversal

The goal of this pass is to reverse the order of iterations in every loop in a given function that satisfies certain conditions. An example is shown in the Figure 3.3 with a loop and its transformation.

3.5.1 Preconditions for Loop Reversal

The following conditions needs to be satisfied for the loop reversal:

- The loop has a single induction variable.

- The induction variable is not modified inside the loop body.

- The induction variable is of integer type.

- The update statement consists of either only adds of a constant parameter or subtractions of a constant parameter.

11

```
       for ( i = a ;  i <= b;  i += c )
2      {
           // statements ;
4      }
```

(a) Original code

```
       for ( i = b−((b−a) mod c ) ;  i >= a ;  i −= c )
2      {
         // modified statements ;
4      }
```

(b) Obfuscated code

Figure 3.3: Loop Reversal

- The value being added or subtracted must remain constant inside the loop.

- Update statements of the form i = i + a; i = i + a + b; i = i - a - b; are allowed.

- Update statements of the form i = i + a - b; i = f(a); i = i * a; i = f(a,i); etc. are not allowed.

- The control never leaves the loop from inside the loop body - there should be no break; or return; statements.

- The condition checked in the loop is a single condition on the induction variable with one of the operators $<, <=, >, >=$.

- Both the lower and upper bounds of the loop must be constant parameters, i.e., their values must not change inside the loop.

- The loop must be in a rotated form - the prerequisite passes *-mem2reg, -loop-simplify, -loop-rotate* must be run.

3.6   Conclusion

Along with the above mentioned techniques we have implemented Function Argument Folding, Loop Splitting, and Constant Encoding. All these passes have been tested for their correctness on LLVM test suite and we ensured that they are bug free.

The strength of the obfuscation is an unsolved problem, in this work, we did not try to give theoretical guaranties, we developed novel techniques and implemented them in LLVM.

Chapter 4

Hardware Circuit Obfuscation

4.1 Introduction

Security of IP in hardware circuits is primary concern for Semiconductor industry. Circuits are reverse engineered to reproduce their functionality. Obfuscation of circuits is the way to protect them from counterfeiting. Various techniques were proposed at hardware level like Design Obfuscation, IP Watermarking, IP Fingerprinting, IC Camouflaging, etc. To the best of our knowledge, none of these techniques are fully secure. We propose to obfuscate the circuit at the software level, using code obfuscation techniques. The LLVM code obfuscation engine we have developed (discussed in the previous chapter) is used to obfuscate the Verilog circuits. In this chapter, we elucidate our approach to this problem and the required components.

4.2 Architecture

The architecture has two flows, first is the circuits that are written in 'C' which are fed to Clang(LLVM Front-end for C/C++) which emits LLVM IR, from there the LLVM Obfucation engine can act on the circuit. We need a Verilog code generator to convert from LLVM IR to Verilog. The other flow is, if the circuit is written in Verilog itself, then a Verilog to C++ translator converts it to C++ which is then fed to Clang and rest of the flow is similar. Figure 4.1 depicts the overall flow.
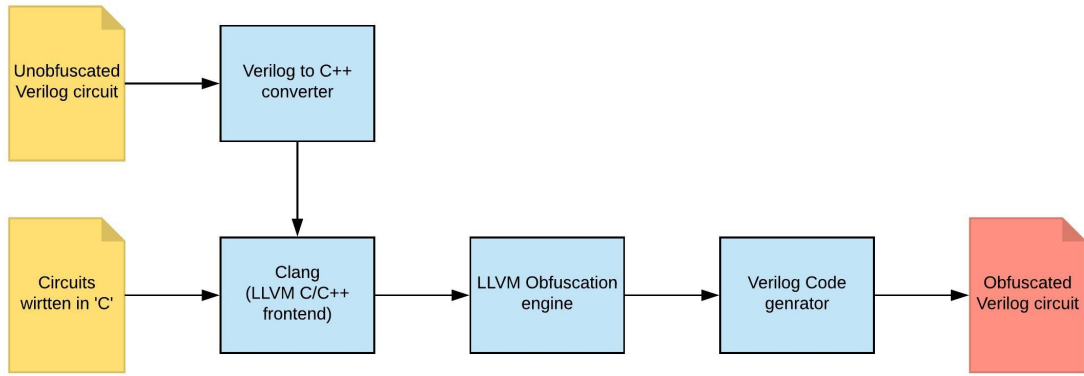
Figure 4.1: Architecture of Hardware obfuscation

4.3 Conclusion

We have procured a Verilog code generator [7] and upgraded it to the version 7.0 which is the latest LLVM version. The *Verilog2C++* [8] converter is an opensource tool which is very old and not been maintained. We have developed the required patches for the *Verilog2C++* to make it usable with the prevalent software. The future work includes development of specific metrics to measures the effectiveness of our approach.

Chapter 5

P4 LLVM Toolchain

5.1   Introduction

Recent research on Software Defined Networking has resulted in the development of programmable data planes. This advancement has made data plane devices re-programmable to suit the requirements of the customers and users after deployment. This flexibility is achieved in-field by configuring these 're-programmable' switches with program configurations. P4 is a high-level imperative programming language for configuring the data plane of the network devices [9]. A compiler translates a P4 program to a switch configuration after performing the necessary optimizations. The front-end of P4LLVM converts the P4-16 code to LLVM-IR. The input P4 code should be checked for lexical, syntactic and semantic correctness before being translated to LLVM-IR. We reuse the open-source P4C front-end module (scanners and parsers) for this purpose. P4 specific passes like removing action parameters, simplifying the key, simplifying the select list, etc. which are the part of mid-end of P4C are also applied after the front-end. The resulting P4 IR is translated to LLVM IR. It is worth noting that all the P4C's passes which have been reused can also be implemented in LLVM. (This is similar to the C/C++ front-ends Clang/Clang++ having their own *front-end passes* independent of the LLVM passes.) Once the P4 code is converted to LLVM IR, any of the target-independent optimizations of LLVM can be applied by a sequence of transformation passes of LLVM.

The code generation to target BMV2 switch involves converting the LLVM IR to JSON. Unlike LLVM IR where every construct is an object, JSON is an open standard file format. So, it is sufficient to generate a JSON format and dump it into a file which could serve as an input to BMV2 switch. This conversion can be conveniently done by an analysis pass of LLVM. The overall flow is summarized in the Fig. 5.1.
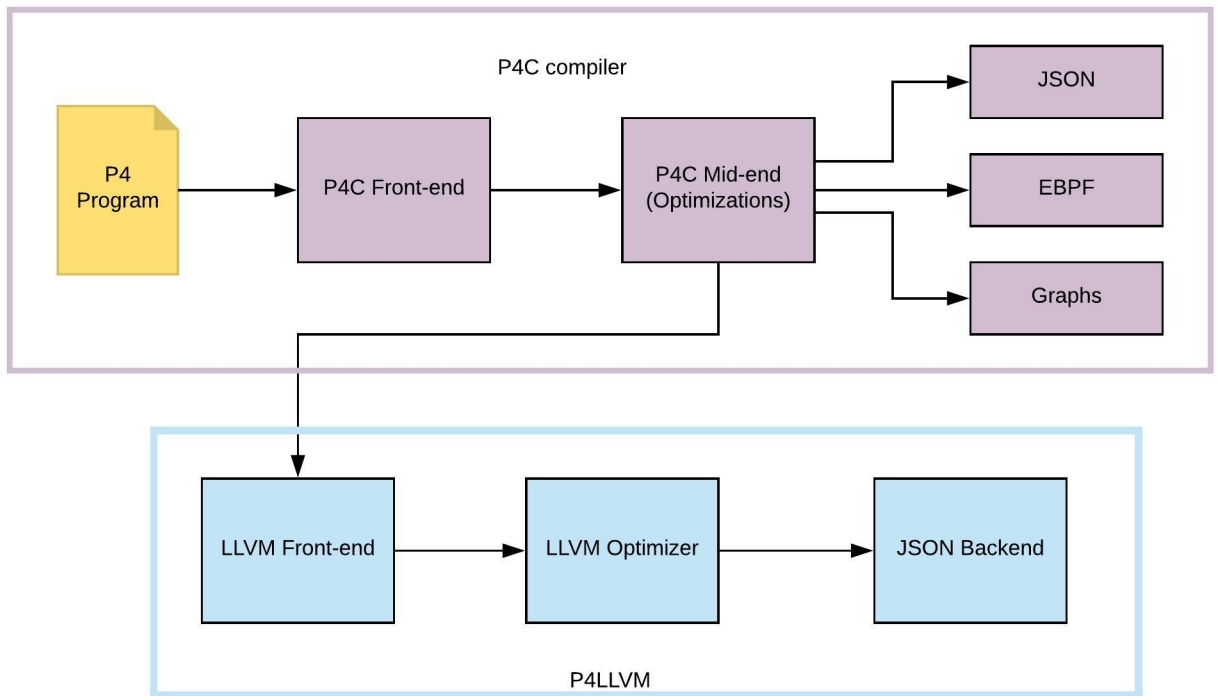
Figure 5.1: P4LLVM Architecture

5.2 Front-end

The front-end of the P4LLVM compiler is responsible for converting P4-IR to LLVM IR, the various P4 constructs and their corresponding LLVM IR constructs are summarized in Table 5.1. Note that there may be more than one way of translation of a construct to its equivalent in another language. So is the case with converting P4 into LLVM IR.

5.3 Back-end

Compilers can generate code for several different architectures. Unlike the general compilers that target CPUs, the P4 compilers target Switch architectures. As of now, Switch architectures have no open standard (like x86, ARM, etc.) for configuring them. With the advent of P4, switch manufacturers are exposing programmability. By using an established compiler framework like LLVM, the already available back-ends can be reused, and the strain of writing back-ends can be avoided. LLVM has code generation support for almost all general purpose CPUs and GPUs like

P4 Construct	Equivalent LLVM IR Construct
Data Types: Headers, Structs	Struct types
Data Types: Header Union	Array of structs
Primitives: int and bit	Int and vector of 1 bit ints
Declarations	Alloca instructions
Assignments	Store instructions
Extern call: extract, verify, setValid/Invalid/isValid and apply	Function prototype and corresponding calls
Tables	Similar to apply
Parser, Control, Action, and Deparser	Functions
Direction in	Passed by value
Direction inout, out	Passed by reference

Table 5.1: Mapping of P4 constructs

NVIDIA and AMD and even has support for accelerators like Hexagon DSP. It is worth mentioning that *p4c-ebpf* compiler translates P4 code to 'C' and then uses *clang(LLVM's C front-end)* for EBPF code generation.

To demonstrate the effectiveness of LLVM optimizations, we have developed a JSON back-end so as to compare with the open-source *p4c-bmv2* compiler. The *"behavioral model(bmv2)"* is a software P4 switch, it can be programmed by a JSON file which could be generated from a P4 compiler. To the best of our knowledge, there are two switch models *portable switch architecture(PSA)* and *v1model*. As of now, *bmv2* has only support for *v1model*. The back-end of P4LLVM translates LLVM IR to JSON format with minimal additional information provided from front-end as metadata. Currently, this back-end can generate code for *v1model*, and it can be extended to have support for other Switch architectures. In LLVM every optimization is a pass over its IR. Because JSON is a simple data representation format, we opted to have JSON emission as an LLVM IR pass. P4LLVM uses the same order of *p4c-bmv2* passes(native P4C optimization passes) to have a comparable JSON code.

Though not extensive, we have tested the correctness of P4LLVM's back-end with the unit test cases provided in the p4c repository [10] as well as with the test cases generated from Whippersnapper [11]. In the current implementation, translation from P4 IR to LLVM IR will not preserve the names of the variables. However, variable names can be recovered if they are attached as metadata in LLVM.

5.4   Results

Whippersnapper is a P4 benchmark suite designed to study the impact of performance caused by compilers. The depth of the Parser tree, Action complexity, Addition/Removal of headers, Table depth in Control blocks are the areas tested by this suite. Dang et al. [11] compare the performance achieved with the existing P4 compilers like P4C [10], P4FPGA [12], PISCES [13] and Xilinx SDNet [14]. We have adopted this work to study the performance impact caused by P4LLVM. Also, we have modified the Whippersnapper tool to generate P4-16 code with complex expressions, conditionals, etc. This modification is essential as P4LLVM, in its current state can support only P4-16 programs. This modified version of the tool, Whippersnapper2.0 [15] is available in GitHub. LLVM has optimization levels O1, O2 and O3 that are more tuned for performance (execution-time), while Os and Oz find a balance between the performance and size of the executable. The size of the P4 program has an impact on packet processing time. Hence, we choose to optimize code at Oz level.
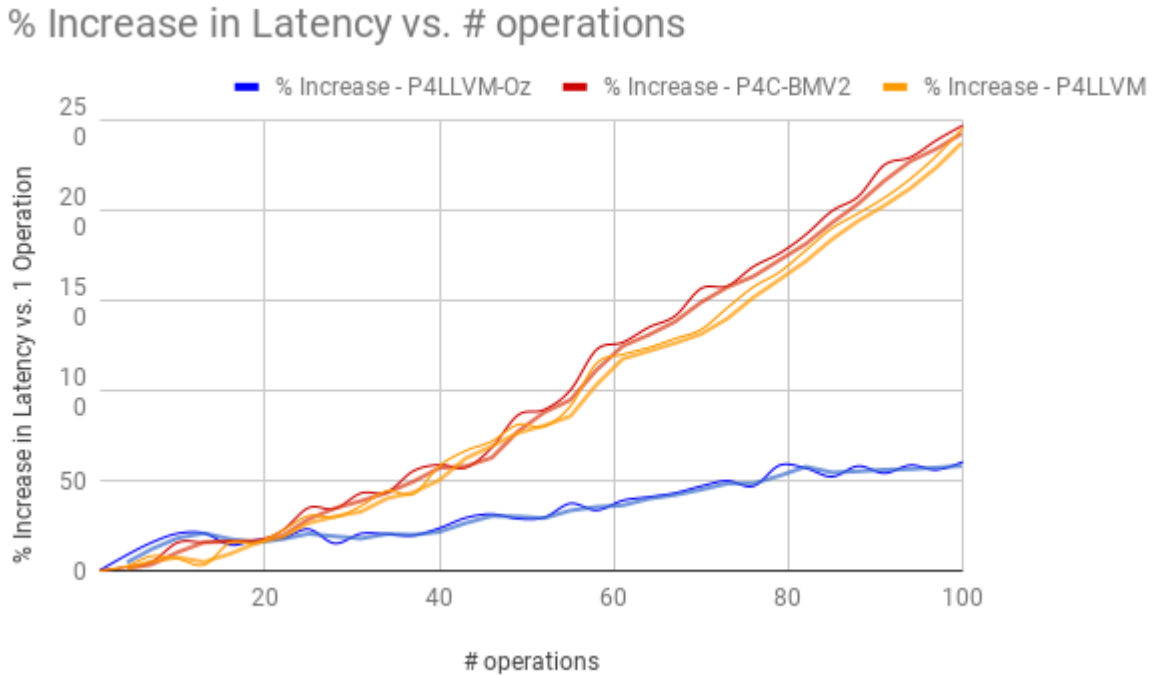

Figure 5.2: Percentage Increase in average latency vs. Number of Operations in the Action block

P4 code is generated using WhipperSnapper2.0 and compiled with P4C, P4LLVM with and without optimization to get the corresponding JSON configurations. This
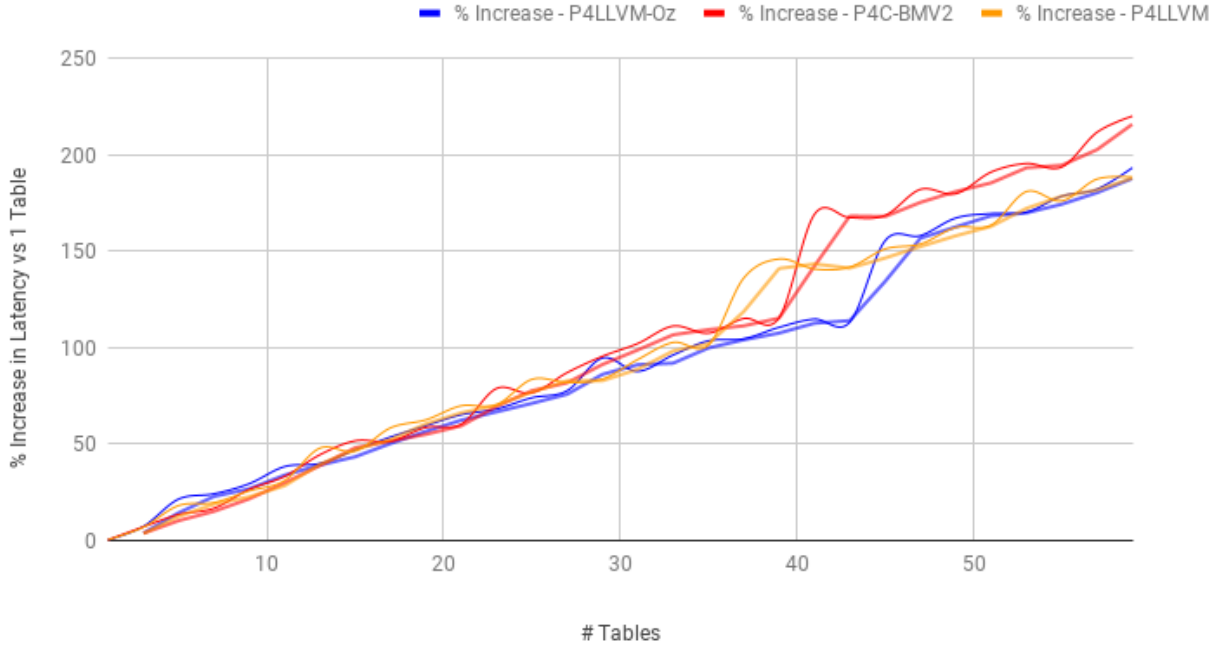
18

Figure 5.3: Percentage Increase in Average Latency vs. Number of tables

is fed to a BMV2 switch and performance is studied. The experiments are performed on a server with 72 cores (dual socket) and Intel Xeon E5-2697 @2.30GHz CPUs, 128GB RAM running Ubuntu 18.04. The latencies in the plots are the average of packet processing time for 10,000 packets.

In Fig. 5.2, we show the increase in average latency versus the number of operations in the Action block. For 100 operations, *p4c-bmv2* compiler registers around 250% increase in latency when compared to 1 operation. Whereas, P4LLVM unoptimized performs similar as *p4c-bmv2* and registers around 245% and the P4LLVM with Oz optimizations registers only 60% increase.

In Fig. 5.3, we show the increase in average latency versus the number of tables; the trend lines show that the performance of P4LLVM-Oz is better than the other two. In both the plots, we notice that performance of P4LLVM (without optimization) is relatively better than P4C, this is because of the minor optimizations done at front-end and back-end code generation. These experiments clearly demonstrate the superiority of P4LLVMs optimization abilities. We believe that with a dedicated optimization sequence designed for P4, the performance would improve further.

Bibliography

[1] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin. Obfuscator-LLVM – Software Protection for the Masses. In B. Wyseur, ed., Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015. IEEE, 2015 3–9.

[2] D. Callahan, J. Dongarra, and D. Levine. Vectorizing Compilers: A Test Suite and Results. Supercomputing '88. IEEE, Los Alamitos, CA, USA, 1988 98–105.

[3] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler Transformations for High-performance Computing. *ACM Comput. Surv.* 26, (1994) 345–420.

[4] D. Nuzman, I. Rosen, and A. Zaks. Auto-vectorization of Interleaved Data for SIMD. In Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '06. ACM, New York, NY, USA, 2006 132–143.

[5] GSoC. PolyhedralInfo - Polly as an analysis pass in LLVM. `http://utpalbora.com/gsoc/2016.html` 2016. [Online; accessed 08-Sep-2017].

[6] A. Darte, Y. Robert, and F. Vivien. Scheduling and Automatic Parallelization. 1st edition. Birkhauser Boston, 2000.

[7] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski. LegUp: High-level Synthesis for FPGA-based Processor/Accelerator Systems. In Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays, FPGA '11. ACM, New York, NY, USA, 2011 33–36.

[8] Verilog2C++. http://verilog2cpp.sourceforge.net/.

[9] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Pro-

gramming Protocol-independent Packet Processors. *SIGCOMM Comput. Commun. Rev.* 44, (2014) 87–95.

[10] M. Budiu, C. Dodd et al. P4C compiler. "https://github.com/p4lang/p4c".

[11] H. T. Dang, H. Wang, T. Jepsen, G. Brebner, C. Kim, J. Rexford, R. Soulé, and H. Weatherspoon. Whippersnapper: A P4 Language Benchmark Suite. In Proceedings of the Symposium on SDN Research, SOSR '17. ACM, New York, NY, USA, 2017 95–101.

[12] H. Wang, R. Soulé, H. T. Dang, K. S. Lee, V. Shrivastav, N. Foster, and H. Weatherspoon. P4FPGA: A Rapid Prototyping Framework for P4. In Proceedings of the Symposium on SDN Research, SOSR '17. ACM, New York, NY, USA, 2017 122–135.

[13] M. Shahbaz, S. Choi, B. Pfaff, C. Kim, N. Feamster, N. McKeown, and J. Rexford. PISCES: A Programmable, Protocol-Independent Software Switch. In Proceedings of the 2016 ACM SIGCOMM Conference, SIGCOMM '16. ACM, New York, NY, USA, 2016 525–538.

[14] SDNet. http://www.xilinx.com/products/design-tools/software-zone/sdnet.html.

[15] Whippersnapper2.0. https://github.com/IITH-Compilers/Whippersnapper-2.0.