Architecture-Aware Optimization Strategies for Instruction Selection in DSP Compilers

Yiwei Wang[1], Jun Wu[2(✉)], Haoqi Ren[3], Zhifeng Zhang[3], and Bin Tan[4]

[1] School of Computer Science and Technology, Tongji University, Shanghai, China
2130778@tongji.edu.cn

[2] School of Computer Science, Fudan University, Shanghai, China
wujun@fudan.edu.cn

[3] School of Electronics and Information Engineering, Tongji University, Shanghai, China
{renhaoqi,zhangzf}@tongji.edu.cn

[4] The College of Electronics and Information Engineering, Jinggangshan University, Ji'an 343009, China
tanbin@jgsu.edu.cn

Abstract. This paper explores the problem of instruction selection optimization in LLVM backend code for transforming platform-independent intermediate code into high-quality target platform instructions. The instruction selection process is divided into two problems: pattern matching and pattern selection. We propose a novel architecture-aware optimization strategy that leverages the features of digital signal processors (DSPs) to improve the efficiency and performance of instruction selection. Our approach involves analyzing the characteristics of DSP architectures to guide the selection of optimal instructions. We evaluate our approach on a set of benchmarks and demonstrate significant improvements in both execution time and code size compared to existing LLVM optimization techniques. Our results show that architecture-aware optimization strategies can effectively enhance instruction selection in DSP compilers, leading to better performance and reduced energy consumption in real-world applications.

Keywords: DSP · LLVM · Instruction-selection · SWIFT · SIMD

1 Introduction

Digital Signal Processor (DSP) is a specialized microprocessor designed specifically for digital signal processing tasks. It analyzes signals in digital form and executes various digital signal processing algorithms in real-time [1]. With the advancement of science and technology, DSP has found extensive applications in multiple domains. LLVM (Low Level Virtual Machine) refers to an open-source collection of modular and reusable compiler and toolchain technologies [2]. LLVM has experienced significant development in recent years and has been adopted in numerous research institutions and commercial projects due to its well-structured modular design, utilization of passes for multi-stage optimization strategies, and high compilation performance.

© ICST Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2024
Published by Springer Nature Switzerland AG 2024. All Rights Reserved

F. Gao et al. (Eds.): Chinacom 2023, LNICST 590, pp. 81–93, 2024.
https://doi.org/10.1007/978-3-031-67162-3_6

Due to the specialized nature of DSP, the LLVM framework is commonly utilized for designing accompanying compilers. Within the compilation process, instruction selection holds significant importance. The process takes an architecture independent intermediate representation called SelectionDAG as input and generate a Directed Acyclic Graph (DAG) consisting of machine code specific to the target platform [3]. This process entails selecting appropriate instructions from the target processor's instruction set to implement the functionalities represented in the intermediate representation. During the design of the instruction selection process, it is crucial to consider the characteristics of different backend architectures and target processor instruction sets. To achieve optimization and generate the most optimal instruction sequences, suitable strategies must be chosen. The quality of the instruction sequence generated through instruction selection significantly impacts program performance and has a substantial influence on the resulting machine code produced by the backend. Therefore, in the instruction selection process of a compiler, designing suitable optimization strategies is essential for fully utilizing the potential of DSP chips and improving program execution efficiency.

The SWIFT DSP is an autonomously designed novel DSP processor that incorporates VLIW (Very Long Instruction Word) and SIMD (Single Instruction, Multiple Data) technologies [4]. It employs a proprietary instruction set and possesses dedicated, flexible, efficient, and highly parallel vector computing capabilities. This paper focuses on the distinctive instruction set of the SWIFT DSP and presents optimization methods for the instruction selection process. Subsequently, the effectiveness of these methods is tested and validated.

The remaining sections of this paper are organized as follows: Sect. 2 discusses the common practices and optimizations employed in the instruction selection process. Section 3 details the optimization methods specifically implemented for instruction selection, specifically for the SWIFT DSP. Section 4 provides an overview of the testing and validation procedures, evaluating the effectiveness of the proposed methods. Finally, Sect. 5 concludes the paper and summarizes the contributions and findings of this research.

2 Related Work

Instruction selection is a crucial aspect of LLVM backend code optimization, aiming to transform platform-independent intermediate code into high-quality target platform instructions [5], effectively mapping IR instructions to machine instructions for the target architecture. The instruction selection process can be divided into the following two problems [6]:

- Pattern matching problem: It involves finding suitable assembly instructions to implement a given segment of IR code. This process takes into account factors such as limitations imposed by the target machine's instruction set, data dependencies within the program, and code reusability.
- Pattern selection problem: It involves determining the combination of assembly instructions to execute the IR code. During this process, considerations need to be given to interactions between instructions, register allocation, code parallelism, and other factors.

In the processes, special considerations need to be given to the influence of DSP architecture characteristics, such as VLIW and SIMD, on instruction selection [7]. Therefore, for DSP compilers, the selection of appropriate optimization strategies for instruction selection is of paramount importance in terms of compiler performance and efficiency.

There are several common optimization techniques in the instruction selection process, including:

- Data flow analysis and register allocation optimization: This approach involves analyzing the data flow and memory access patterns of the program to design optimization methods [8]. For instance, in the TI C64x+ DSP architecture, a technique called "Dataflow Analysis Optimization" is employed. By analyzing the data flow and usage, it optimizes the execution order of instructions and register allocation, thereby enhancing program efficiency.
- Peephole optimization [9]: This technique focuses on local optimizations of small code sequences during code generation to improve code performance and efficiency [10]. In ARM Cortex-M series processors, a method called "Peephole-based Instruction Selection" is employed. It identifies recurring instruction sequences and patterns, optimizing them to enhance program efficiency.

The above-mentioned methods are general optimization techniques [11]. In this paper, we focus on the unique instruction set of SWIFT DSP. By conducting analysis of the internal architecture and instruction set characteristics of the target DSP architecture, we aim to optimize instruction selection by considering specific patterns in instruction sequences. This approach aims to enhance performance and reduce resource consumption in the instruction selection process during code generation.

3 Optimization Strategies

The SWIFT DSP is an innovative digital signal processor developed independently by Tongji University. It features a vector instruction set that supports SIMD capabilities, utilizes a nine-stage pipeline structure and employs VLIW processing technology. It features eight parallel execution units, known as slots, which significantly enhance program execution speed. This paper focuses on the instruction selection method based on SelectionDAG and primarily analyzes the vector instructions in the proprietary instruction set of SWIFT DSP. The aim is to design optimization strategies for the instruction selection process.

3.1 Instruction Selection Algorithm

SelectionDAG is an implementation of trees-on-DAGs, and tree patterns representing instructions are written and translated into complete tree-matching code using tablgen. This code is then processed by the matcher generator, which employs a greedy DAG-to-DAG strategy to match the target-independent intermediate representation (IR) instructions to machine instruction descriptions. The IR is equivalent to low-level assembly instructions, providing an infinite number of virtual registers and support for commonly

used instruction sets. The steps of instruction selection in SelectionDAG involve designing a target-specific machine function pass, which operates by traversing all basic blocks within a function. The specific steps are illustrated in the algorithm below (Fig. 1):

Algorithm 1 LLVM Intermediate Expression to SelectionDAG Conversion

```
1: Input: LLVM intermediate expression
2: Output: Target-independent SelectionDAG
3: procedure ConstructInitialDAG
4:     Convert LLVM instructions to SelectionDAG nodes
5:     Implement special instructions using target-specific functions
6: end procedure
7: procedure LegalizeTypesAndOperators
8:     Legalize operand types and operators
9: end procedure
10: procedure InstructionMatching
11:     Transform target-independent DAG nodes to target-specific nodes
12:     Implement instruction selection algorithm
13: end procedure
14: procedure InstructionScheduling
15:     Schedule instructions in DAG
16:     Assign linear order to instructions based on dependencies and constraints
17: end procedure
18: procedure MainProcedure
19:     ConstructInitialDAG
20:     LegalizeTypesAndOperators
21:     InstructionMatching
22:     InstructionScheduling
23: end procedure
```

Fig. 1. Pseudocode Presentation of LLVM Instruction Selection Process

LLVM offers a range of standard ISD (Instruction Selection DAG) intermediate instructions that align with their corresponding C language programs for general programs. The design of scalar instructions is mostly similar across different target platforms. In contrast, vector instructions exhibit significant variations due to differences in chip designs. As an illustration, the following are a few SWIFT DSP-specific vector instructions.

1) vector comparison instruction (VGT, VLT, VEQ, VGE, VLT)

   This type of instruction is used for comparing vector data. It performs a comparison between the vector elements from two vector input registers and stores the comparison results in the corresponding groups of the destination vector register. The comparison can include operations such as greater than, less than, or equal to.

2) VMOVC.V2V

This instruction performs a vector data select transfer, using the result of the afore-mentioned vector comparison instruction as one of the input data. It determines whether the data should be transferred to the destination register based on the condition value obtained from the comparison.

3) VMAX/VMIN

This instruction is used for finding the maximum value in vector data. It performs a signed comparison between the groups of data from two input vector registers and stores the maximum value in the corresponding group of the destination register.

These instructions cannot be directly mapped to the standard intermediate instructions in LLVM. They either correspond to a combination of several intermediate instructions or have functionalities that cannot be directly represented by the existing intermediate instructions. Therefore, in this paper, these instructions are collectively referred to as "high-level instructions". If the original complex intermediate instructions are not optimized during instruction selection, they may indirectly match into multiple scalar instructions, failing to fully utilize the parallelism of vector instructions and exploit the hardware advantages. In the following sections, a method for optimizing instruction selection is proposed to match suitable ISD intermediate instructions to the afore-mentioned high-level instructions, enabling the full utilization of vector instruction parallelism and improving performance.

3.2 Optimized Instruction Selection Algorithm

In LLVM, the instruction selection process employs a tree pattern matching algorithm that utilizes a work queue to store the pending instruction selection subtrees. This algorithm retrieves the smallest subtree from the queue and uses the OpCode of its root node and the operand types of its leaf nodes to invoke the matching function in the SelectionDAGBuilder class, and finally replaces the smallest subtree with a machine instruction node. As a result, the smallest subtree is reduced to a leaf node and then reintegrated into the original DAG. The algorithm terminates when the work queue is empty.

Based on the preceding analysis, an architecture-specific instruction selection optimization is performed on SelectionDAG during its traversal. The algorithm is as follows, and it is essential to note that this optimization relies on the prerequisite that the corresponding vector types have been legalized (Fig. 2).

Algorithm 2 Instruction Selection Optimization

```
1: Input: Platform-independent SelectionDAG
2: Output: Platform-specific MachineDAG
3: Procedure INSTRUCTIONSELECTIONOPTIMIZATION(SeletionDAG)
4: for each node Node in SeletionDAG do
5:   if Node corresponds to a vector instruction then
6:     continue to the next node
7:   end if
8:   if a match is found for a high-level instruction then
9:     Replace the original instruction sequence with a high-level instruction
       sequence
10:  end if
11: end for
12: return the optimized MachineDAG
13: End Procedure
```

Fig. 2. Pseudocode Presentation of Instruction Selection Algorithm

3.3 Optimization in the Examples

Algorithm 2 serves as the foundation for the optimizations proposed in this paper. In the actual instruction selection process, the details of the algorithm need to be adjusted based on different high-level instructions. The following provides specific explanations regarding the example presented in Sect. 3.1.

1) vector comparison instruction (VGT, VLT, VEQ, VGE, VLT)

   These instructions compute comparison results between vectors. They take two input vectors of the same type and produce comparison results that are distributed bit-wise among the respective vector parts (VPs). The corresponding ISD instruction is a vector form of SETCC and can be optimized and selected as a vector comparison instruction. It should be noted that the result of this SETCC operation, after computation, consists of densely packed '0' or '1' values on a per-bit basis, rather than following the standard vector format of SWIFT. Compared to the algorithm framework described in Sect. 3.2, in this case, only a specific portion of the algorithm undergoes more detailed processing. The specific algorithm is outlined as Fig. 3.

   This optimization combines the setcc instructions into a single vector comparison instruction. It also serves as a foundational step for optimizing the subsequent vselect instructions. Additionally, after this optimization, the comparison results need to be extended to the appropriate format based on the data alignment standards of the corresponding architecture.

Algorithm 3 Instruction Selection Optimization

1: **Input**: Platform-independent SelectionDAG
2: **Output**: Platform-specific MachineDAG
3: **Procedure** VectorComparisonOptimization(*Node*)
4: **for each** *Node* in SelectionDAG **do**
5: ...
6: **if** *Node*'s opcode is ISD SETCC and both compared operands are vector types **then**
7: Match the appropriate vector comparison instruction for *Node* based on the condition code and operand vector types
8: For the condition code "!=" combine VEQ and VNOT instructions for matching
9: **end if**
10: **if** *Node* has subsequent zero-extending references **then**
11: Convert the bitwise-packed comparison results to the SWIFT standard format, assigning a bool value to each integer vector element
12: **end if**
13: **end for**
14: **return** the replaced MachineDAG
15: **End Procedure**

Fig. 3. Pseudocode Adjustment for Vector Comparison Instructions

2) VMOVC.V2V

This instruction is typically used in combination with the aforementioned comparison instructions and the select intermediate instruction. It is used to process two vector data sets, compare them, and based on the comparison results, select one set of values for output. The corresponding ISD instruction is a combination of SETCC and SELECT. The comparison performed by SETCC follows the algorithm outlined in Algorithm 3, and the result is combined with SELECT to choose the VMOVC.V2V instruction. The specific algorithm is as Fig. 4.

Without this optimization, vselect would be selected as a combination of scalar select instructions for the correspond-ding data groups. After the optimization, it is selected as the VMOVC.V2V instruction, effectively utilizing parallel computing capabilities.

Algorithm 4 Instruction Selection Optimization

Require: Platform-independent SelectionDAG
Ensure: Platform-specific MachineDAG
1: **Procedure** VMOVCV2V OPTIMIZATION(*SelectionDAG*)
2: **for each** *Node* in *SelectionDAG* **do**
3: ...
4: **if** *Node*'s opcode is ISD SETCC and the next instruction is a SELECT instruction **then**
5: Replace *Node* with a vector comparison instruction
6: Select the comparison result and the next node as a VMOVCV2V instruction
7: Choose one vector to correspond to the original value of the comparison result and the other to the negation of the comparison result
8: Apply the VMOVCV2V instruction to the two sets of values and add them to obtain the final result
9: **end if**
10: **end for**
11: **return** the optimized MachineDAG
12: **End Procedure**

Fig. 4. Pseudocode Adjustment for VMOVC.V2V Instruction

3) VMAX/VMIN

The optimization of the VMAX/VMIN instructions builds upon Algorithm 4 mentioned above. If the two vectors used for comparison in Algorithm 4 are the same as the two vectors from which the final selection is made, there is no need to use the vector comparison instruction. Instead, SETCC and SELECT can be optimized into a single VMAX instruction, which selects the maximum or minimum value between the two vectors. The corresponding ISD instruction is a combination of SETCC and SELECT, and the computation process can be represented as follows: CmpResult = SetCC(A, B), Result = Select(CmpResult, A, B). The process is similar to the aforementioned methods, so this paper will not provide a detailed algorithm. This optimization further improves the two specific cases mentioned above by combining the vselect instructions and comparison instructions of the same data group into VMAX/VMIN instructions, enhancing the execution speed in such scenarios.

4 Evaluation

Since instruction selection is only an intermediate step in the code generation process of a compiler, there are subsequent modules such as register allocation and instruction scheduling that can affect the final code generation. As instruction selection can alter the instruction sequence, it cannot guarantee the consistency of the subsequent modules. Therefore, DAG verification is adopted to demonstrate the optimization results of instruction selection, with the assistance of the number of program execution cycles to showcase the optimization outcomes. The optimization results for the aforementioned example are presented below.

Graphviz is a graph visualization tool that allows for the representation of structural information as abstract graphs and networks, using the dot tool to visualize the graphs [13]. In our research, we utilize Graphviz to visualize the control flow graph of LLVM IR, enabling us to gain a clear understanding of the node information and the flow of code execution. By providing an LLVM IR intermediate code file, we can utilize the Graphviz tool to construct DAG graphs for different stages of the backend code generation process.

4.1 Vector Comparisons and VMOVCV2V

We present the optimization results for vector comparison and VMOVCV2V together, and below are partial DAG graphs showing the before and after effects of instruction selection optimization. From the graphs, it can be observed that the vselect and setcc nodes prior to the instruction selection optimization are replaced by a combination of vgt comparison instructions and VMOVCV2V instructions after optimization. Without this optimization, the vselect instruction would require separate scalar select operations for each group of data in the vector, significantly increasing the number of execution cycles. However, through optimization, parallel instructions can be utilized, resulting in improved efficiency (Figs. 5 and 6).

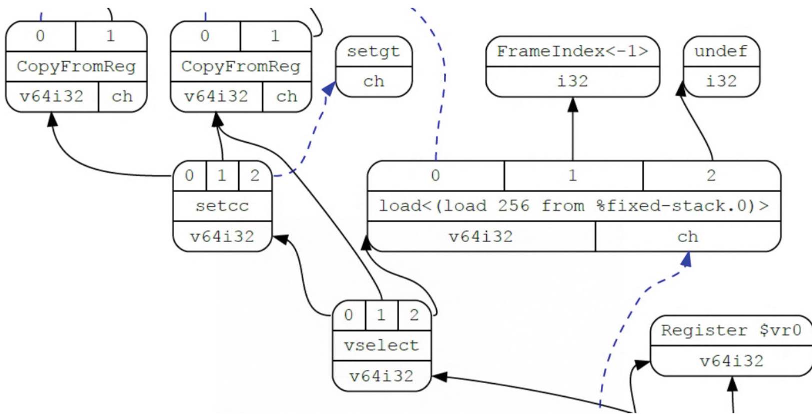

Fig. 5. Partial DAG before Instruction Selection Optimization

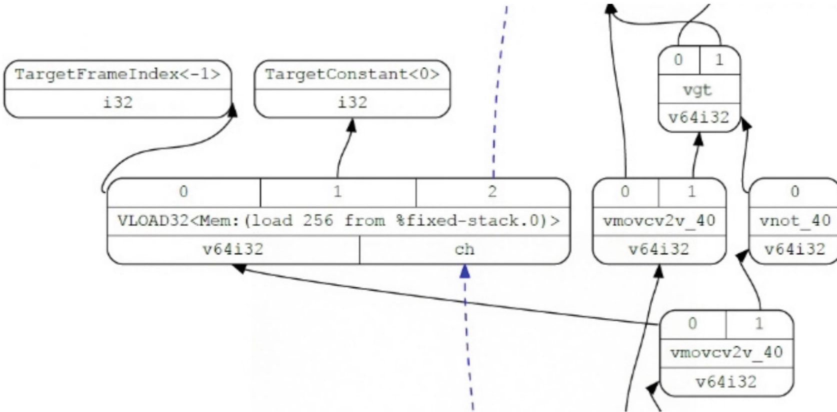

Fig. 6. Partial DAG after Instruction Selection Optimizations

4.2 VMAX/VMIN

The following shows the DAG graph for instruction selection optimization of VMAX/VMIN. From the graph, it can be seen that the vselect and setgt nodes before optimization require conditional checks followed by data selection based on the comparison results. After optimization, these operations are consolidated into a single VMAX instruction, reducing code length and complexity, and improving program execution efficiency (Figs. 7 and 8).

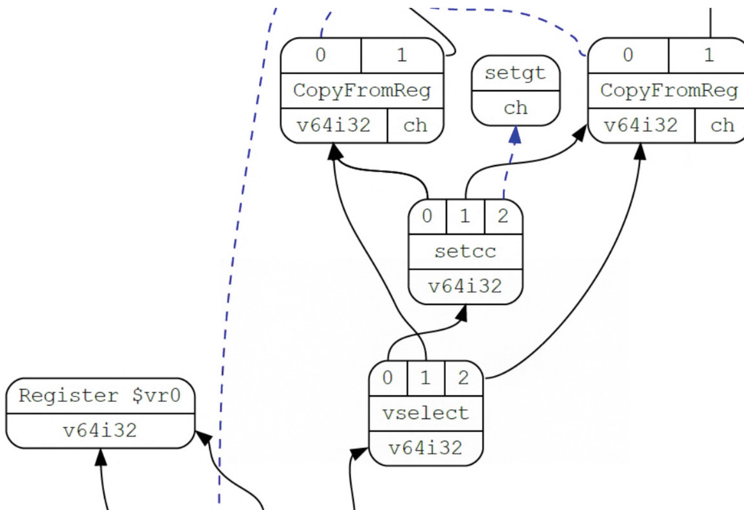

Fig. 7. Partial DAG before Instruction Selection Optimization

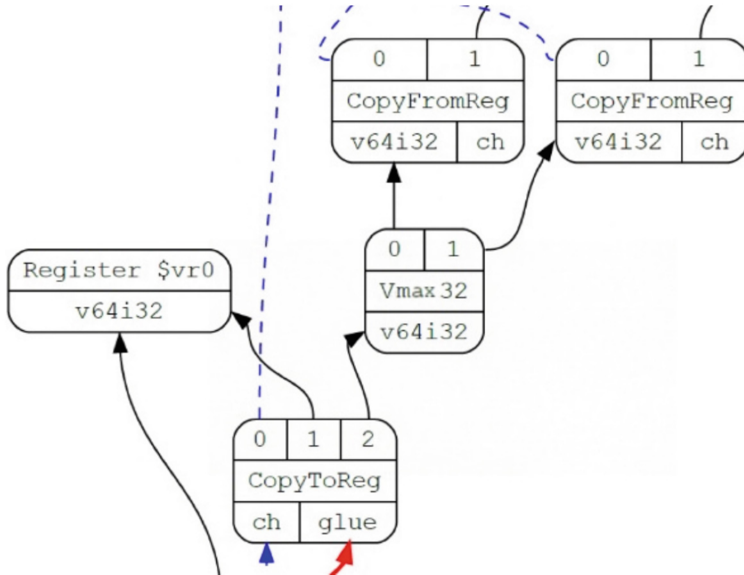

Fig. 8. Partial DAG before Instruction Selection Optimization

The Table 1 presents a comparison of average cycle counts before and after optimization for ten SWIFT test programs that involve the optimizations. Vselect IR is selected as scalar select instructions corresponding to vector grouping before optimization. After optimization, it is selected as the vmovc2v instruction. In the second case, before optimization, it is selected as a combination of comparison instruction and vselect instruction. After optimization, it is selected as a VMAX/VMIN instruction. The important part of the test program is shown in the Fig. 9 and Fig. 10.

```
define <64 x i32> @vector_select( <64 x i32> %vec1, <64 x i32> %vec2_1, <64 x
i32> %vec2_2) {
    %compare_result = icmp sgt <64 x i32> %vec1, %vec2_1
    %result = select <64 x i1> %compare_result, <64 x i32> %vec2_1, <64 x
i32> %vec2_2
    ret <64 x i32> %result
 }
```

Fig. 9. Key code example of the optimization test program for Vselect

```
define <64 x i32> @vector_smax(<64 x i32> %vec1, <64 x i32> %vec2) {
    %cmp = icmp sgt <64 x i32> %vec1, %vec2
    %result = select <64 x i1> %cmp, <64 x i32> %vec1, <64 x i32> %vec2
    ret <64 x i32> %result
}
```

Fig. 10. Key code example of the optimization test program for Vmax

Table 1. Demonstration of Optimization Results

Table column subhead	Before optimization	After optimization
Vector Comparisons And VMOVCV2	1135.5	934
VMAX/VMIN	975.3	869.6

5 Conclusion

In this paper, we have presented an architecture-aware optimization strategy for instruction selection in LLVM backend code optimization. Our approach leverages the unique features of SWFIT DSP to improve the efficiency and performance of instruction selection. We have shown that DSP-specific characteristics can guide the selection of optimal instructions and lead to significant improvements in both execution time and code size. In addition, there are other vector instructions available for optimization, such as VABS, VMUL, VSUM, VMAC and so on. This paper did not provide a detailed analysis of them, but further optimization and improvement can be explored in future work.

Our evaluation results demonstrate that our approach outperforms existing LLVM optimization techniques on a set of benchmarks, and our approach can be extended to other types of processors beyond SWFIT DSP. By analyzing the unique features of different processor architectures, we can develop architecture-aware optimization strategies that are tailored to specific platforms. This could lead to significant improvements in performance and energy efficiency for a wide range of applications.

In addition, our work highlights the importance of considering the characteristics of the target platform when designing optimization strategies for LLVM backend code. By taking into account platform-specific features, we can develop more effective optimization techniques that fully utilize the capabilities of modern processors. We hope that our research will inspire further exploration in this area and lead to new breakthroughs in compiler optimization.

References

1. Eyre, J., Bier, J.: The evolution of DSP processors. IEEE Signal Process. Mag. **17**(2), 43–51 (2000)
2. Lattner, C., Adve, V.: The LLVM compiler framework and infrastructure tutorial. In: Eigenmann, R., Li, Z., Midkiff, S.P. (eds.) LCPC 2004. LNCS, vol. 3602, pp. 15–16. Springer, Heidelberg (2005). https://doi.org/10.1007/11532378_2
3. Lattner, C., Adve, V.: LLVM: a compilation framework for lifelong program analysis & transformation. In: International Symposium on Code Generation and Optimization (CGO), San Jose, pp. 75–86. IEEE (2004)
4. Haoqi, R., Zhifeng, Z., Jun, W.: A VLIW DSP for communication applications. In: 2015 Sixth International Green Computing Conference and Sustainable Computing Conference (IGSC), pp. 1–5. IEEE (2015)
5. Guobin, Y.E.: Getting to know the LLVM compiler. Master's thesis, The University of Edinburgh (2011)
6. Pandey, M., Sarda, S.: LLVM Cookbook. Packt Publishing Ltd. (2015)

7. Keith, D., Cooper, L.T.: Engineering a Compiler. Posts Telecom Press (2020)
8. Lorenz, M., Leupers, R., Marwedel, P.: Low-Energy DSP Code Generation Using a Genetic Algorithm. IEEE (2001)
9. Lee, J., Hur, C.-K., Lopes, N.P.: AliveInLean: a verified LLVM peephole optimization verifier. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11562, pp. 445–455. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25543-5_25
10. Mullen, E., Zuniga, D., Tatlock, Z., et al.: Verified peephole optimizations for CompCert. In: ACMSIGPLAN Notices, vol. 51, no. 6, pp. 448–461. ACM (2016)
11. Wulf,W., et al.: The Design of an Optimizing Compiler (1975)
12. Joshi, R., Nelson, G., Zhou, Y.: Denali: a practical algorithm for generating optimal code. ACM Trans. Program. Lang. Syst. **28**(6), 967–989 (2006). https://doi.org/10.1145/1186632.1186633
13. Erhardt, C., Scheler, D.I.F.: Design and implementation of a TriCore backend for the LLVM compiler framework. Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU) (2010)