Optimization based on LLVM global instruction selection

Huang Zhufeng\(^1\), Shang Jiandong\(^2\)

\(^1\) Department of Information Engineering, Zhengzhou University, Zhengzhou, Henan Province, 450000, China

\(^2\) Henan Supercomputer Center, Zhengzhou University, Zhengzhou, Henan Province, 450000, China

\(^*\) Corresponding author’s e-mail: hzhufeng@outlook.com

Abstract. Instruction selection is a key component of code generation. High-quality instruction selection has a great impact on the size and quality of the generated code. The existing instruction selection technology is mostly limited to a single statement or a single basic block, and the global instruction selection based on LLVM degrades the entire function in the form of SSA. Global instruction selection optimization based on LLVM is implemented on Shenwei platform, including global instruction merge optimization based on cost model, register bank selection optimization and instruction locality optimization. Through the test of SPEC CPU2006, Experimental results show that the average speed-up ratio before and after the optimization of global instruction selection based on LLVM is 1.08, and the maximum speed-up ratio is 1.36. In addition, when the quality of the generated code is equivalent, the global instruction selection is compared with the default instruction selection, the LLC compilation speed is increased by an average of 20%, and the entire compilation cycle is increased by an average of 6%-8%.

1. Research Background

LLVM\(^1\) is an emerging lightweight open source compilation system. Its core library provides support for compilation tools and can be used as a backend for multiple language compilers. LLVM provides compile-time optimization, link-time optimization, runtime optimization, and idle optimization of high-level language programs. LLVM adopts the intermediate representation in the form of SSA\(^2\). LLVM-IR provides a virtual instruction set independent of the target platform. The instruction set is a three-address instruction set similar to the simplified instruction set, including simple control instructions, calculation instructions and access Store instructions. Currently, the LLVM backend supports the generation of codes for mainstream processors such as X86, Sparc, PowerPC, Mips, MVPTX, ARM, and AMDGPU. The entire LLVM compilation system is structurally divided into three parts as shown in Figure 1. The front end is responsible for parsing, diagnosing and verifying the input high-level language code, and then converting the high-level language to LLVM-IR; the middle end performs a series of analysis on LLVM-IR And optimization passes to improve the quality of the generated code; the back-end downgrades LLVM-IR to a binary file executable on the target platform.
In recent years, the development and compilation optimization research based on the LLVM compilation system is very active, the most famous of which is the LLVM Global Developers Conference held in early April and early October every year [3]. The back end of the LLVM compiler consists of instruction selection, instruction scheduling and register allocation, and some optimizations. Instruction selection [4] The goal is to find an effective mapping from the target platform-independent intermediate representation to the target platform-related assembly. It is a key component of code generation and has a great impact on the size and quality of the generated code [5.6.7.8.9.10.11.12.13]. The classic instruction selection methods include macro expansion, tree coverage, Directed Acyclic Graph (DAG), etc. They limit the selection range to statements or basic blocks, and only achieve local code optimization. DAG matching is an NP-complete problem [14]. While instruction selection achieves local code optimization, it also brings a lot of compilation overhead. In 1999, Ertl [15] proposed a tree matching method based on DAG, which can determine whether it can provide the best instruction selection result for the specified sentence. Global Instruction Selection (GlobalISel) [16.17] is the theme of the LLVM Global Developers Conference in October 2019. It not only solves the compilation overhead of DAG-based instruction selection due to the introduction of DAG, but also solves the problem of DAG-based instruction selection cannot be degraded and optimized across basic blocks.

2. Based on LLVM global instruction selection
Global instruction selection is a framework that provides a collection of reusable optimization modules and instruction selection tools. It directly converts LLVM-IR into machine instructions for the target platform. The goal of global instruction selection is to replace DAG-based instruction selection and fast instruction selection, and mainly solve the following three problems.

1. Compilation performance issues: SelectionDAGISel introduces a new DAG intermediate representation, sacrificing compilation time. GlobalISel directly generates target machine instructions for LLVM-IR operations.

2. Optimize the granularity problem: Some global optimization opportunities will be lost on the basic blocks of SelectionDAGISel and FastISel operations. While GlobalISel operates on the entire function, it will retain complete LLVM-IR information and explore more global optimization opportunities.

3. Modularity issues: SelectionDAGISel and FastISel are both single modules, with less shared code and poor reusability. GlobalISel is created in a way of code reuse, LLVM shares their compilation pipeline, and the target platform can configure the pipeline to better meet their needs.

The architecture of global instruction selection is shown in Figure 2 below. The basic global instruction selection framework includes four modules, namely IRTranslator, Legalizer, RegisterBankSelector, and InstructionSelector. IRTranslator acts on LLVM-IR and converts LLVM-IR to Generic Machine IR (GMIR). Legalizer converts the operand types of general machine instructions.
not supported by the target platform to the operand types of the target machine instructions supported
by the target platform. The legalization module is also responsible for converting operations that are
not supported by the target platform. RegisterBankSelector allocates a register bank for the operand
of each mixed machine instruction. InstructionSelector converts mixed machine instructions into target
machine instructions. Generally, the quality of the generated code after the four modules of the basic
instruction selection framework is often low. Therefore, the global instruction selection also provides
a combination of instructions that can be configured for the target platform (optional, flexible).

| LLVM-IR | Generic Machine Instructions (gMIR) | Generic Machine Instructions and Machine Instructions (gMIR and MIR) | Machine Instructions (MIR) |
|---------|-------------------------------------|---------------------------------------------------------------|---------------------------|
| IR Translator | Legalizer | Register Bank Selector | Instruction Selector |
| IR Translator | Combine1 | Legalizer | Combine2 | Register Bank Selector | Combine3 | Instruction Selector | Combine4 |

Fig. 2 GlobalISel lowering process

3. Optimization based on global command selection

3.1 Instruction merging based on cost model

The process of instruction merging is shown in Figure 3 below. The original instruction merging instance was limited to a single instruction, and processed the special operands of a single instruction. For complex merging of multiple instructions, non-TableGen custom merging rules need to be established to merge instructions. For example, floating-point multiply and add instructions are merged into multiply-add instructions, floating-point multiply and subtract instructions are merged into multiply-subtract instructions, and so on. There may be expansion operations between instructions when multiple instructions are merged. Direct merging of radical instructions may not be beneficial. Therefore, a precise instruction cost model is added to the original instruction merge. The instruction cost model analyzes before and after the instruction merge. The sum of the instruction cost of the order, the order is merged only when it is judged that there is a profit. The abstract algorithm of instruction merging is an abstraction of all non-TableGen custom merging rules. The implementation of each instruction merging is specific, and there is a big deviation in practice, but the calculation of the instruction cost is the same, both are the sum of the instruction costs to be merged and the sum of the instruction costs generated after the merge.

Fig. 3 The process of instruction combine

3.2 Global register bank selection optimization

Register selection optimization analyzes the cost of transferring data across registers during register
group selection, and minimizes the overhead of transferring data across registers between instructions.
The optimization of global register bank selection extends the optimization of register bank selection
and uses more global analysis, for example, analyzing the frequency of machine basic block operation and the branch probability of machine basic block, and it does some along the chain of instruction definition and use in the form of SSA Radical optimization, such as transferring data across register banks. The global register set allocation algorithm traverses the basic blocks in the function in a reverse post-sequence traversal manner, and then allocates the register set to the instructions in each basic block in a sequential manner. It first analyzes which registers can be used for each operation of the target instruction Group and what the cost is, and then choose to minimize the instruction cost to copy the general virtual register operands to the correct register bank. The cost of a GMI instruction is calculated as shown in Formula 1. The cost of instruction I on a certain register set is equal to the cost of I operating on the register set plus the sum of the cost of copying the operand from the current register set to another register set.

$$\text{cost}(I, \text{RegBank}) = \text{cost}(I, \text{Opcode}, \text{RegBank}) + \sum(\text{for each in } I.\text{args}: \text{costCrosscopy}(\text{arg, RegBank}, \text{RegBank}))$$

(1)

### 3.3 Instruction locality optimization

Instruction locality optimization can effectively solve the problem of register overflow in the register allocation stage by reducing the active range of variables. In the process of downgrading LLVM-IR to GMIR, constants or some variables are instantiated in the function entry block. Because register allocation cannot re-instantiate constants, it has to deal with a longer active range of variables, which may be possible under certain physical registers. Cause register overflow. Instruction partial optimization re-instantiates constants in places close to use, creating smaller active intervals. The realization of instruction locality optimization, first collect the variables that need to be re-instantiated in the function entry block, and determine which basic block to instantiate it in; secondly, re-instantiate the variable through the copy instruction and insert it into the corresponding instance to be instantiated in the basic block of the variable; finally, the second positioning in the basic block, through the move instruction to make the definition of the variable closer to use, reduce the active range of the variable, to solve the problem of register overflow in the register allocation stage. The instruction locality optimization algorithm is shown in Figure 4 below.
Instruction local optimization

4. Experiment and analysis
The experimental platform uses the Shenwei 1261 processor; the LLVM compiler version is Release 10.0; the test set uses SPEC2006[18]. SPEC2016 test scale ref.LLVM compiler options: The default optimization level is O0-O3, and the global instruction selects the compiler option -fglobal-isel.

Table 1. Optimization of experimental data based on global instruction selection of LLVM

| Program        | Type | Compilation speedup | running speedup | optimized before/after |
|---------------|------|----------------------|----------------|------------------------|
|               |      | O3       | O2       | O1       | O0       | O3       |                |                        |
| 400.perlbench | C/INT | 1.207    | 1.198    | 1.227    | 0.903    | 0.987    | 1.07          |                        |
| 401.bzip2     | C/INT | 1.200    | 1.187    | 1.206    | 0.952    | 0.986    | 1.06          |                        |
| 403.gcc       | C/INT | 1.181    | 1.2      | 1.2      | 0.937    | 0.980    | 1.01          |                        |
| 429.mcf       | C/INT | 1.210    | 1.199    | 1.186    | 1.027    | 0.990    | 1.01          |                        |
| 445.gobmk     | C/INT | 1.197    | 1.208    | 1.206    | 1.156    | 0.985    | 1.04          |                        |
| 456.hmmer     | C/INT | 1.214    | 1.217    | 1.205    | 0.933    | 0.988    | 1.1           |                        |
| 458.sjeng     | C/INT | 1.203    | 1.195    | 1.235    | 0.952    | 0.983    | 1.09          |                        |
| 462.libquantum| C/INT | 1.232    | 1.260    | 1.225    | 0.978    | 0.984    | 1.06          |                        |
| 464.h264ref   | C/INT | 1.187    | 1.180    | 1.18     | 0.922    | 0.986    | 1.11          |                        |
| 471.omnetpp   | C++/INT | 1.203   | 1.207    | 1.202    | 0.898    | 0.978    | 1.05          |                        |
| 473.astar     | C++/INT | 1.190   | 1.194    | 1.188    | 1.040    | 0.984    | 1.08          |                        |
| 483.xalancbmk | C++/INT | 1.190   | 1.203    | 1.193    | 0.904    | 0.989    | 1.04          |                        |
| 433.milc      | C/FP  | 1.214    | 1.230    | 1.230    | 0.888    | 1.031    | 1.02          |                        |
| 444.namd      | C++/FP | 1.188   | 1.191    | 1.174    | 0.940    | 0.989    | 1.11          |                        |
| 447.dealll    | C++/FP | 1.202   | 1.187    | 1.198    | 0.964    | 1.007    | 1.09          |                        |
Global instruction selection and DAG-based instruction selection compilation speed are shown by the absolute compilation time of the LLC compilation stage. The compilation time of LLC phase accounts for about 30%-40% of the entire compilation cycle. The experimental test results of the compilation speed are shown in columns 3, 4, 5, and 6 in Table 1. The data in each column represents the speedup ratio of the program compilation time under different optimization levels. Global instruction selection and DAG-based instruction selection code generation quality is expressed by the absolute running time of the program. The experimental test results of code generation quality are shown in the 7 columns in Table 1. The 7th column represents the ratio of the running time of the generated code based on the global instruction selection and the DAG-based instruction selection at the O3 optimization level.

From the experimental results of compilation speed and code generation quality, it can be seen that when optimization is turned on under the Shenwei platform, global-based instruction selection and DAG-based instruction selection are equivalent to the generated code quality (on average, only a 1% reduction- 2%), the average compilation time is increased by 20%. When optimization is not turned on, the average compilation time of global-based instruction selection is reduced by about 4% compared with fast instruction selection, and the quality of code generation is improved. The last column in the table represents the acceleration ratio before and after the addition of global instruction selection optimization. It can be seen from the data in this column that the optimization based on global instruction selection has an acceleration effect on most programs, with the average acceleration ratio of 1.08 and the highest acceleration ratio of 1.36.

5. conclusion
Global-based instruction selection is the newly introduced instruction selection framework of the LLVM compiler, which not only solves the problem that fast instruction selection does not support illegal data types and operations, but also solves the loss of global optimization ability caused by the introduction of DAG based on DAG instruction selection, and The compile-time overhead caused by DAG. Global instruction selection optimization based on LLVM is implemented on Shenwei platform. When the optimization is turned on, the compilation time of LLC is increased by 20%, and the entire compilation cycle from high-level language to executable program is increased by 6%-8% under the premise that the quality of the generated code is equivalent. Realize optimization based on LLVM global instruction selection for Shenwei platform, which improves code generation quality by an average of 8%, but the quality of generated code is still worse than DAG-based instruction selection. The next step is to further analyze the DAG stage node fusion and global instruction selection stage The similarities and differences of instruction merging, more examples of instruction merging are introduced to further improve the performance of the compiler.

References
[1] Lattner C ,  Adve V . LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation[C]// Code Generation and Optimization, 2004. CGO 2004. International Symposium on. 2004.
[2] Appel, Andrew W . SSA is Functional Programming[J]. Acm Sigplan Notices, 1998.
[3] LLVM Developer Meetings Website. https://llvm.org/devmtg.
[4] Blindell H , Gabriel. Instruction Selection[J]. 2016, 10.1007/978-3-319-34019-7.
[5] Ertl M A ,  Casey K ,  Gregg D . Fast and flexible instruction selection with on-demand tree-parsing automata[C]// ACM, 2006:52.
[6] Guo J ,  Limberg T ,  Matus E , et al. Code Generation for STA Architecture[J]. 2006.
[7] João Dias, Ramsey N. Converting Intermediate Code to Assembly Code Using Declarative Machine Descriptions[C]// Springer Berlin Heidelberg, 2006.
[8] Eckstein E, Knig O, Scholz B. Code Instruction Selection Based on SSA-Graphs[C]// Software and Compilers for Embedded Systems, 7th International Workshop, SCOPES 2003, Vienna, Austria, September 24-26, 2003, Proceedings. 2003.
[9] Leupers R, Bashford S. Graph based Code Selection Techniques for Embedded Processors[J]. Acm Transactions on Design Automation of Electronic Systems, 2000.
[10] Ebner D, Brandner F, Scholz B, et al. Generalized instruction selection using SSA-graphs[J]. Acm Sigplan Notices, 2008, 43(7).
[11] Nymeyer A, Katoen J P. Code generation based on formal BURS theory and heuristic search[J]. Acta Informatica, 1997, 34(8):597-635.
[12] Liao S, Devadas S, Keutzer K, et al. Instruction selection using binate covering for code sizeoptimization[C]// Computer-Aided Design, 1995. ICCAD-95. Digest of Technical Papers. 1995 IEEE/ACM International Conference on. ACM, 1995.
[13] Fraser C W, Henry R R, Proebsting T A. Burg --- Fast Optimal Instruction Selection and Tree Parsing[J]. Acm Sigplan Notices, 1992, 27(4):68-76.
[14] Todd A. Proebsting. Least-Cost Instruction Selection in DAGs is NPComplete. http://research.microsoft.com/~toddpro/papers/proof.htm, 1998.
[15] M. Anton Ertl. Optimal Code Selection in DAGs. In Principles of Programming Languages (POPL '99), 1999.
[16] Daniel Sanders, Aditya Nandakumar, Justin Bogner. Tutorial: Head First into GlobalISel. LLVMDevMeeting 2017.
[17] Volkan Keles, Daniel Sanders. Generating Optimized Code with GlobalISel. LLVMDevMeeting 2019.
[18] SPEC2006 Website. http://www.spec.org