Optimization of misaligned data processing based on LLVM compiler

Lingqin Gong[a, *], Qinglei Zhou[b] and Hao Hu[c]

School of Information Engineering, Zhengzhou University, Zhengzhou 450000, China

[a]youxiangxl01@sina.com, [b]ieqlzhou@zzu.edu.cn, [c]witstorm @163.com

*Corresponding author

Keywords: LLVM, Directed acyclic graph, misaligned data, Instruction selection

Abstract: The processing of misaligned data is optimized based on the directed acyclic graph at the back end of the LLVM compiler. Implements the LLVM compiler's processing of unaligned scalar data and unaligned vector data. The processing and optimization of misaligned data mainly includes two aspects: 1. Provides optimized processing of misaligned vector data and scalar data based on directed acyclic graph, which solves the problem of misaligned memory access affecting program performance. 2. In the LLVM compiler backend instruction selection phase, a custom node downgrade process is provided, which enhances the flexibility and reusability of the optimization. Test the program containing misaligned data. The maximum speedup of the problem with unaligned scalar data is 20.27, the average speedup is 14.49, the maximum speedup of the problem with unaligned vector data is 14.01, and the average speedup is 13.61.

1. Introduction

LLVM (Low Level Virtual Machine) is a widely used compiler framework developed by the University of Illinois [1]. LLVM provides an intermediate representation LLVM IR (Intermediate Representation) based on SSA (Static Single Assignment) [2].

Memory address alignment requires that when reading and writing data, the destination address must be an integer multiple of the number of bytes of the basic data type accessed [3]. On the processor side, memory alignment requires that the memory addresses accessed by the processor when reading and writing data are aligned [4]. Access aligned memory addresses if and only if A mod n = 0, where A is the memory address and n is the width of the fetched data in bytes [5, 6]. When a fetch operation is an unaligned fetch, the value of a mod n determines the offset of the unaligned address relative to the aligned fetch address.

A directed acyclic graph is a loopless directed graph. In a DAG, for any node N, there is no directed path that starts at N and ends at N [7]. DAG is an important data structure in the computer field. Due to its unique topology, DAG are widely used in a variety of algorithm scenarios such as dynamic planning, shortest path finding, and data compression [8]. In the LLVM compiler, the backend uses a DAG to visualize the backend downgrade process [9, 10]. Instruction selection is an important stage of backend degradation. The SelectionDAG generated during the instruction selection process is a directed acyclic graph. The node SDNode in the SelectionDAG is the node that carries the IR operation or operand [11].

The instruction selection through the DAG is a process of converting the LLVM IR into a SelectionDAG node (SDNode) of the target instruction. Compiler optimization processing is divided into target platform-dependent optimization and target platform-independent optimization. Target platform-independent optimization has the advantage of being versatile and not constrained by specific platform characteristics. In order to fully exploit the optimization potential of the compiler and enhance the generality of the optimization, in the instruction selection stage, a custom downgrade processing of the DAG (Directed Acyclic Graph) node is implemented to achieve target independence for accessing misaligned data addresses Optimization.

2. Misaligned data processing strategy

2.1 Custom degradation of vector data DAG nodes

A single instruction multiple data (SIMD) processor can perform the same processing on multiple data at the same time, and SIMD can efficiently perform a large amount of data level parallelism (DLP) processing [12]. SIMD fetch instructions can load or store multiple data items with one instruction [13]. By adding the corresponding SIMD memory access instruction in the instruction set, the data parallelism in the program is fully utilized, and the performance improvement is achieved. In practical problems, when the memory addresses of vector data are not aligned, the vectorization of the program will be affected, resulting in a decrease in SIMD performance [14].

During the process of custom downgrading the load and store nodes of vector data, data offset nodes, data splicing operation nodes, misaligned fetch nodes and data flow nodes are used to simulate load or store operations. The data offset node is responsible for obtaining and storing the offset information of the misaligned data in order to extract and write back valid data from the memory unit. The data splicing operation node is responsible for splicing the data to obtain complete valid data. The new load and new store nodes are custom load and store nodes after degraded processing, and can perform misaligned fetch operations on processed data. After the custom downgrade processing is completed, the nodes generated by the downgrade continue the downgrade process at the back end and finally generate assembly instructions. The custom degradation process of the load node of vector data is shown in Figure 1.

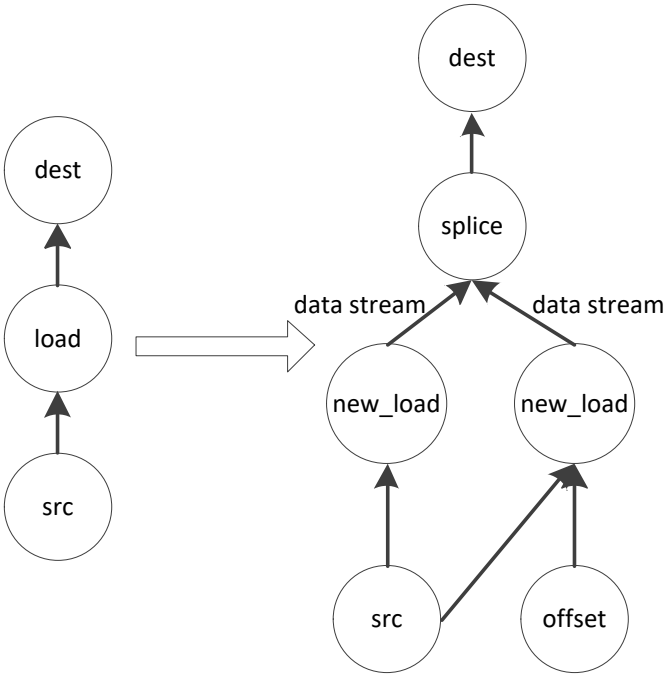

Figure 1. Vector load node custom degradation processing

When the vector data type is downgraded, the original load node is downgraded to multiple nodes, and multiple nodes generated by the downgrade are processed for implicit misaligned data. The src node represents the source register of the load operation, and dest represents the destination register of the load operation. The offset node represents the offset of valid data in the memory unit. The splice node represents a data splicing operation node. This node splices the loaded data to obtain valid data. The data stream represents the data stream nodes in the back end of the compiler, and the data stream nodes maintain the data dependency relationship between the custom DAG nodes. After the load node of the vector data is subjected to custom downgrade processing, the new load node loads the data from the starting address and starting address of the memory unit plus the offset memory through two loads. The data stream node passes the data to the data splicing node

splice. The data splicing node splices the data and passes the complete and valid data obtained to the superior DAG node.

2.2 Custom degradation of scalar data DAG nodes

The custom degradation processing phase of the scalar data DAG node is the same as the custom degradation processing phase of the vector data. When the intermediate representation of LLVM starts to downgrade through the instruction selection stage based on the DAG, by performing a custom downgrade process on the load node and the store node, the custom nodes in the DAG are used to complete the loading and storage of misaligned data. The degradation process of the load node of scalar data is shown in Figure 2.

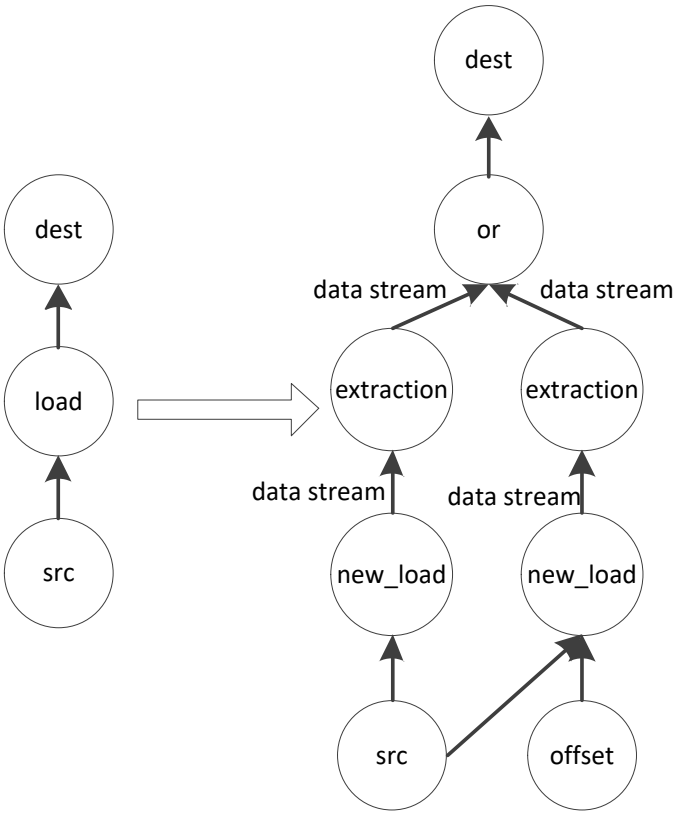

Figure 2. Custom downgrade processing for scalar load nodes

When the scalar data type is downgraded, the original load node is downgraded to multiple target-independent nodes. The compiler performs implicit misaligned data processing by degrading multiple nodes. The extraction node represents a data extraction node. This node extracts the valid data in the memory unit and extracts the valid data bits into an all-zero memory area to ensure that the memory unit where the valid data is located has no dirty data bits. The OR node represents a logical OR node. This node performs a bitwise OR operation on the data bits in the memory unit of the two extraction operations to obtain complete valid data. After the load node of the scalar data undergoes a custom downgrade process, the new load node loads the data from the starting address and starting address of the memory unit plus the offset address through two loads. The data flow node passes data to the data extraction node, and the data extraction node performs extraction operations on the data loaded twice and passes the extracted data to the logic OR node through the data flow node. The logical OR node performs a bitwise logical OR operation on the two incoming operands to obtain complete valid data and passes the valid data to the superior DAG node.

3. Implementation of misaligned data processing technology

3.1 Vector data misaligned processing implementation

Taking the load operation of unaligned vector data as an example, the degradation algorithm of the custom load node of unaligned vector data is shown in Algorithm 1.

Algorithm 1: custom degradation processing of misaligned vector load nodes

```
Input: src, offset
Output: dest
If Op. getValueType () is vector then
  Case Op.getOpcode () in
Load)
      If Op.getAlignment () is false then
      low←creatNode (src, 0)
      offset←creatNode (src, offset)
      low_data←valChain (low)
      offset_data←valChain (offset)
      dest←splice (low_data, offset_data)
    Else
     Lower Load ()
    End if
Break
End case
End if
```

Algorithm one is the pseudo-code for custom downgrade processing of the vector load node. The algorithm treats the nodes in SelectionDAG as an operation Op and downgrades the nodes. The algorithm uses the getValueType method to obtain the data type and determines whether the data type is a vector. If the data type is a vector, a custom downgrade processing flow of the node of the vector data type is performed. The algorithm uses the getOpcode method to obtain the operation code of the operation, and matches the operation code of the node through the case. If the node is a load node, it uses the get Alignment method to determine whether the node is a load operation for processing unaligned data. If the data is aligned data, no custom downgrade processing is performed, and the load node is naturally downgraded. If the data is misaligned data, two new load nodes are created by the cretonne method and two loading operations are performed from the memory unit according to the address and offset value. The loaded data is transmitted to the superior DAG node through the data stream node. After the data splicing node splice receives the data transmitted by the data stream node, it performs the splicing operation on the valid data to obtain the complete valid data.

3.2 Scalar data misaligned processing implementation

Take the load operation of misaligned scalar data as an example. The degradation algorithm of the custom load node of misaligned scalar data is shown in Algorithm 2.

Algorithm 2: custom degraded processing of unaligned scalar load nodes

```
Input: src, offset
Output: dest
If Op. getValueType () is scalar then
Case Op. getOpcode () in
Load)
        If Op.getAlignment () is false then
        low←creatNode (src, 0)
        offset←creatNode (src, offset)
        low_data←valChain (low)
        offset_data←valChain (offset)
```

```
        eff_low ←extraction (low_data)
      eff_offset ←extraction (offset_data)
      low_val ← valChain (eff_low)
      offset_val ← valChain (eff_offset)
      dest←or (low_val, offset_val)
    Else
      Lower Load ()
    End if
  Break
  End case
  End if
```

Algorithm two is pseudo-code for custom downgrade processing of scalar load nodes. The algorithm uses the getValueType method to obtain the data type and determines whether the data type is a scalar. If the data type is scalar, the custom downgrade processing flow of the node with the scalar data type is performed. The algorithm uses case statements to match node operation types through node operation codes. If the node is a load node, it is determined whether the node processes misaligned data. A natural degradation process is performed on the load node that processes the aligned data. If the data is misaligned, create two new load nodes and perform two load operations from the memory unit based on the address and offset values. Pass the loaded data to the data extraction node through the data flow node. The data extraction node extracts part of the valid data loaded twice into different all-zero memory areas, and passes the extracted data to the logic or node through the data flow node. Logic or node performs bitwise OR operation on valid data to get complete valid data.

4. Results

4.1 Unaligned scalar data test

Scalar data types include char, short, int, long, float, and double. Data address misalignment is closely related to platform hardware characteristics [15]. For sunway processor, the minimum granularity of aligned access is four bytes. Addresses of 32-bit data and 64-bit data types need not be misaligned. LLVM compilers for sunway processor need misaligned data processing to optimize data types including 8-bit and 16-bit data types char and short.

Compile the following test program and run it on sunway processor. In the program, TYPE1={int, long, double, float}, TYPE2={char, short, int, float}.

```
Volatile TYPE1 b;
TYPE2 a []={1,2,3,4,5,6};
Void main () {
For (int i=0; i<1000000000ULL; i++) {
      b = *(TYPE1 *) (a+1);
}
}
```

Through the comparison experiments on the optimization of unaligned data processing and the optimization of unaligned processing, the optimization effect of unaligned scalar data processing is verified. The maximum speedup of the test case is 20.27, and the average speedup is 14.49. The experimental results are shown in Table 1

Table 1. Test results of unaligned scalar data

TYPE1	TYPE2	Time before optimization (s)	Time after optimization (s)	Speedup ratio
double	char	187.20	16.18	11.57
double	float	187.20	16.18	11.57
double	int	187.91	16.19	11.56
double	short	187.20	16.17	11.58
float	char	199.31	15.72	12.68
float	short	197.90	15.80	12.53
int	char	182.30	15.80	11.54
int	short	182.12	15.75	11.56
long	char	183.16	9.70	18.88
long	float	182.89	9.12	20.05
long	int	183.05	9.05	20.23
long	short	183.06	9.03	20.27
short	char	180.91	12.56	14.40
		Average acceleration ratio		14.49

The experimental results show that when billions of accesses are made to the misaligned data, the maximum acceleration ratio for processing unaligned scalar data is 20.27, and the average acceleration ratio is 14.49.

4.2 Unaligned vector data test

The vector data types that need to be misaligned in the sunway processor and the LLVM compiler include v4f64, v4f32, and v8i32. Compile the following test program and run it on the sunway processor. In the program TYPE1= {v4f64, v4f32, v8i32}, TYPE2={int, long, float, double}.

```
Volatile TYPE1 b;
TYPE2 a [100];
Void main () {
   For (int i=0; i<1000000000ULL; i++) {
     b= *(TYPE1 *) (a + 1);
}
}
```

Through the comparison experiments on the optimization of misaligned data processing and the optimization of misaligned processing, the optimization effect of the vector data processing is verified. The test case has a maximum speedup of 14.01 and an average speed up of 13.61. The experimental results are shown in Table 2.

Table 2. Unaligned vector data test results

TYPE1	TYPE2	Time before optimization (s)	Time after optimization (s)	Speedup ratio
v4f64	long	199.90	14.64	13.65
v4f64	double	199.91	14.66	13.64
v4f32	float	197.20	14.61	13.50
v4f32	int	197.14	14.64	13.46
v4f32	long	194.59	14.63	13.30
v4f32	double	194.67	14.66	13.29
v8i32	float	205.24	14.65	14.01
v8i32	double	199.70	14.62	13.66
v8i32	long	199.90	14.68	13.62
v8i32	int	204.64	14.63	13.98
		Average acceleration ratio		13.61

The experimental results show that when billions of accesses are made to unaligned data, the maximum speedup of unaligned vector data processing is 14.01 and the average speedup is 13.61.

5. Conclusion

This article describes the optimization of misaligned data processing based on directed acyclic graphs. Based on the LLVM compiler, it provides a solution for custom downgrading of nodes during the instruction selection stage, and optimizes the load and store operations of misaligned scalar data types and misaligned vector data types. Based on the DAG, use the custom node downgrade process on the LLVM compiler backend to handle load and store nodes for unaligned data.

In the test case of unaligned scalar data, the maximum speedup is 20.27 and the average speedup is 14.49. In the test case of unaligned vector data, the maximum speedup is 14.01 and the average speedup is 13.61. Through the implementation of custom degraded processing optimization for unaligned data in this paper, the LLVM compiler can further utilize the program's data parallelism to improve program performance. The next work is to further study the vector data misaligned data processing and fully exploit the parallel computing capabilities of sunway processor.

References

[1] Lam, Siu Kwan, Antoine Pitrou, and Stanley Seibert. "Numba: An llvm-based python jit compiler." Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC. ACM, 2015.

[2] Sui, Yulei, and Jingling Xue. "SVF: interprocedural static value-flow analysis in LLVM." Proceedings of the 25th international conference on compiler construction. ACM, 2016.

[3] Ringe, Tushar P., et al. "Data processing apparatus with snoop request address alignment and snoop response time alignment." U.S. Patent Application No. 10/042, 766.

[4] Turner, Andrew Edmund, George PATSILARAS, and Bohuslav Rychlik. "Cache line compaction of compressed data segments." U.S. Patent No. 10,261,910. 16 Apr. 2019.

[5] Han, Woojong, et al. "Apparatus and method to support a storage mode over a cache-line memory interface to a non-volatile memory dual in line memory module." U.S. Patent No. 10,067,879. 4 Sep. 2018.

[6] Liu, Shaoli, et al. "Cambricon: An instruction set architecture for neural networks." ACM SIGARCH Computer Architecture News. Vol. 44. No. 3. IEEE Press, 2016.

[7] Vasseur, Jean-Philippe, et al. "Dynamic directed acyclic graph (DAG) adjustment." U.S. Patent No. 8,489,765. 16 Jul. 2013.

[8] Wang, Ling, and Hichem Sahbi. "Directed acyclic graph kernels for action recognition." Proceedings of the IEEE International Conference on Computer Vision. 2013.

[9] Antao, Samuel F., et al. "Offloading support for OpenMP in Clang and LLVM." Proceedings of the Third Workshop on LLVM Compiler Infrastructure in HPC. IEEE Press, 2016.

[10] Lee, Juneyoung, et al. "Taming undefined behavior in LLVM." ACM SIGPLAN Notices. Vol. 52. No. 6. ACM, 2017.

[11] Horváth, Gábor, and Norbert Pataki. "Clang matchers for verified usage of the C++ Standard Template Library." Annales Mathematicae ET Informaticae. Vol. 44. 2015.

[12] Ross, Scott. "Systems and methods for using alternate computer instruction sets." U.S. Patent No. 10,007, 520. 26 Jun. 2018.

[13] Polychroniou, Orestis, Arun Raghavan, and Kenneth A. Ross. "Rethinking SIMD vectorization for in-memory databases." Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. ACM, 2015.

[14] Mantor, Michael J., and Brian Emberling. "SIMD processing unit with local data share and access to a global data share of a GPU." U.S. Patent No. 9,619,428. 11 Apr. 2017.

[15] Plotnikov, Mikhail, and Igor Ermolaev. "Instruction set for eliminating misaligned memory accesses during processing of an array having misaligned data rows." U.S. Patent No. 9,910,670. 6 Mar. 2018.