Research and Implementation of LLVM JIT Porting for Sunway Processor

Jiandong Shang[1,a], Hongsheng Wang[1,b] and Mengyao Chen[1,c]

[1]School of information Engineering, Zhengzhou University, Zhengzhou, China
a. shangjiandong@hotmail.com, b. whs1814@foxmail.com, c. cmy564088954@163.com

Keywords: LLVM, JIT compilation, sunway processor, porting.

Abstract: Sunway processor is a general-purpose processor with independent instruction set and complete independent intellectual property rights, and has been successfully applied in various fields. As an open source compilation framework, LLVM has been widely adopted by commercial and open source projects. It is of great practical significance to implement the porting of LLVM to Sunway platform. Through the analysis and research of the basic composition of the JIT (just-in-time) compilation system and the functional principles of each component in LLVM, combined with the backend porting mechanism of the open source compiler, the porting of the JIT compiler on the Sunway multi-core processor platform is implemented. Relevant tests were completed based on the LLVM test-suite, and the correctness of the transplanted LLVM JIT compiler was verified.

1. Introduction

The LLVM project was initiated by the University of Illinois to provide a modern compilation research framework based on SSA (static single assignment), which can support static and dynamic compilation of any programming language[1]. At present, LLVM has developed into an overall project composed of multiple sub-projects, which is a collection of modular, reusable compilers and tool chain technologies.

LLVM is now used as a general infrastructure to implement a wide variety of static and runtime compiled languages[2]. Generally, a compiler implementation only provides traditional static compilers (such as GCC, Free Pascal, and Free BASIC), or a runtime compiler in the form of an interpreter or just-in-time (JIT) compiler. The LLVM compiler framework not only supports both static compilation and runtime compilation, but the two share most of the code.

The LLVM JIT compiler is a function-based dynamic transformation engine[3]. The JIT compiler does not store program binaries on disk, but compiles programs on demand at runtime instead. An important feature of the JIT compilation strategy is that during the compilation of the program, the JIT system can accurately understand the type of platform on which the program will run and the underlying core architecture of the platform, which is conducive to compiling the program code into a more efficient and suitable executable running on the platform. In addition, in

some cases, the compiler is used to compile part of the code only when the program is running, in which case it is extremely necessary to implement and deploy a JIT compilation system. For example, when a GPU program is running, any GPU intermediate code loaded by the program will be further compiled into device-dependent binary executable code by the graphics card driver.

Due to the advantages of high modularity and strong optimization performance of the LLVM compiler framework itself, more and more projects use LLVM as a compilation tool and are widely used in academic research. By converting JavaScript language source code into LLVM IR (intermediate representation) code and then optimizing it by heavyweight, Apple increased the speed of the Safari JavaScript engine Nitro by 35%[4]. At the same time, the just-in-time compilation technology in LLVM has also achieved quite important optimization results in many practical engineering applications. Michael Larabel et al. Compile SQL queries through JIT in PostgreSQL, avoiding passing SQL queries through the Postgres interpreter, so in database tests such as TPC-H, the speed of compiling expressions for PostgreSQL is more than 20% faster, and the speed of index creation It can even increase by 5-19%[5]. Yu-Hsin Tsai et al. modified the JNA (Java Native Access) source code and integrated the LLVM JIT compiler into JNA to improve performance. Their experiments achieved a performance improvement of approximately 8% to 16% when calling native functions with different types and numbers of parameters[6]. Michal Gregor et al. Used LLVM-based JIT compilation in genetic programming, focusing on executing evolutionary programs as fast as possible, improving the computational efficiency of genetic programming, and reducing the total execution time[7].

With the rapid development of domestic processors, it is very important to build basic software such as the software ecosystem and compilation toolchain of domestic platforms, which is conducive to the gradual improvement of the ease of use and robustness of domestic processors and related supporting tools and software. In addition to developing software with independent intellectual property rights, it is also of great significance to absorb and learn from excellent open source software with wide application. The Sunway platform supports LLVM JIT compilation, which is conducive to improving the software ecology of the Sunway platform, and is conducive to the migration of many LLVM-based business projects and application software to the Sunway platform, which promotes the IT system to be autonomous and controllable. The porting work of this article is based on LLVM 3.3, and the backend target is Sunway multi-core processor.

2. LLVM Just-in-time Compilation

There are two types of JIT compilation frameworks (JIT and MCJIT) in LLVM 3.3. LLVM has supported JIT since the release of version 1.0, and the MCJIT framework based on MC (Machine Code) layer support has been introduced since version 2.9. In the version 3.3, the JIT framework has been very mature, and the MCJIT framework has just been introduced. The relevant interfaces and features are not mature and perfect, for example, there is no support for GOT. And, a lot of software that uses the LLVM JIT system still uses the JIT framework. JIT and MCJIT are two completely different implementations. This paper will mainly discuss the mature JIT framework.

2.1.JIT Framework

The JIT framework is implemented by using various parts of the LLVM code generator. The entire JIT framework is shown in Figure 1. It includes common target-independent function modules, which mainly include JIT execution engine, JIT event listener, target-independent code emitter and JIT memory manager, and target-related function modules, mainly target machines, Code generator and target JIT support interface. Each functional module is implemented by related classes.

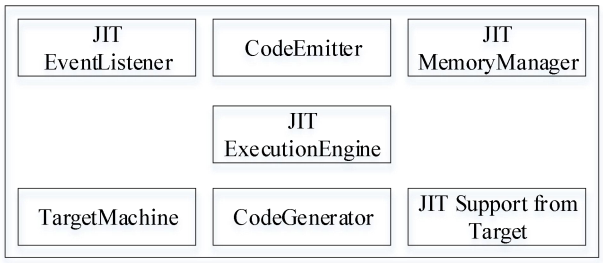

Figure 1: JIT framework.

2.1.1. JIT Execution Engine

The JIT execution engine mainly depends on six other functional modules: JITMemoryManager implements JIT memory management, JITCodeEmitter implements target-independent code emission interfaces, TargetJITInfo is a public interface supported by a specific target JIT, TargetMachine represents the target machine, JITEventListener is an event listener, and JITState As a code generator. The JIT engine works by compiling and executing LLVM IR functions at runtime. During the compilation phase, the JIT engine uses the LLVM code generator to generate target-specific binary instructions. Returns a pointer to a compiled function, and the function can be executed. The JIT engine is almost independent of targets, but each target must implement binary instruction emission for its specific instruction.

2.1.2. JIT Memory Management

In just-in-time compilation, memory management is essential for routine tasks such as memory space allocation, release, library loading, and memory permission processing [3]. The JIT execution engine uses the DefaultJITMemoryManager class for memory management by default, which is derived from the RTDyldMemoryManager base class. When this class is initialized, it will apply to the operating system for 512kb of memory space for storing JIT-compiled binary instructions, and another 512kb of space for function stubs. This class mainly implements the methods to complete the following tasks: resolve the address of the available symbols in the current linked library, apply for new memory space from the operating system when the requested free memory space is insufficient, and allocate the available RWX Memory block for the compiled function binary code, which allocates space for auxiliary structures such as function stubs, global variables, exception tables, and GOT tables, and sets memory permissions on allocated memory blocks. Any JIT client can also provide a custom RTDyldMemoryManager subclass to specify where different JIT components should be placed in memory.

2.1.3. JIT Event Listener

JITEventListener is an abstract interface used by JIT to notify clients of important events during compilation. For example, tell the profiler and debugger where the function was emitted and release the memory occupied by the specified function. The default implementation of each method does nothing and requires setting specific parameters when using LLVM to use JIT event listeners.

2.1.4. Target-independent Code Emission

The JIT execution engine uses JITCodeEmitter to issue binary instructions. The JITCodeEmitter class is a target-independent code emission interface that is called by the target-related code

emission interface during the code emission process. This class mainly implements relatively simple methods such as writing bytes to memory, for example, writing binary machine instructions to a memory buffer, tracking the current buffer address, and so on. The more complex tasks are implemented by JITEmitter, the subclass of JITCodeEmitter. JITEmitter implements various complex auxiliary operations for code emission. JITEmitter mainly completes the following tasks: allocate space for the current function to be emitted, release the memory space occupied by the specified function, and issue auxiliary structures for the function (such as constant pool, jump table, Relocation, etc.), and writing GOT tables. In addition, JITEmitter has a dedicated memory manager JITMemoryManager relative to JITCodeEmitter, as well as a parser instance to keep track of and resolve call points to functions that have not yet been compiled, which is essential for lazy function compilation.

2.1.5. Target Machine Description

TargetMachine is a complete description interface of the target machine, through which all target-specific information can be accessed. The target-unrelated part interacts with the target machine through this interface, for example, to obtain the target processor's data layout, instruction description, register description and other information, add target-specific compilation Pass in the code generator, set target options, and so on. Therefore, all backend target machines need to inherit the TargetMachine base class.

2.1.6. Code Generator

The code generator is mainly composed of a set of passes of compiled functions. All the passes are executed by the run method of FunctionPassManagerImpl. Each pass analyzes or converts one function at a time, and converts LLVM IR into machine-dependent MachineInstr step by step. Finally, the machine-related object code emitter converts MachineInstr into machine binary instructions, and then uses the target-independent code emission interface to send the binary instructions into memory.

2.1.7. JIT Support

The TargetJITInfo class provides an interface for the JIT functions that each target needs to implement. For example, application relocation, emission function stubs, lazy parsing, function recompilation, etc. Each target needs to implement a TargetJITInfo subclass for JIT support.

2.2.Implementation Mechanism of JIT Compilation

The LLVM JIT compiler compiles one function at a time, which defines the granularity of compilation, that is, LLVM uses function-based just-in-time compilation technology. The JIT framework supports lazy-compilation. By compiling functions on demand, the JIT system will only compile and execute functions that are actually used in the program. For example, if a program has multiple functions but provides incorrect command line arguments when starting it, a function-based JIT system will only compile functions that output help messages, not the entire program. The process of JIT lazy compilation is shown in Figure 2.

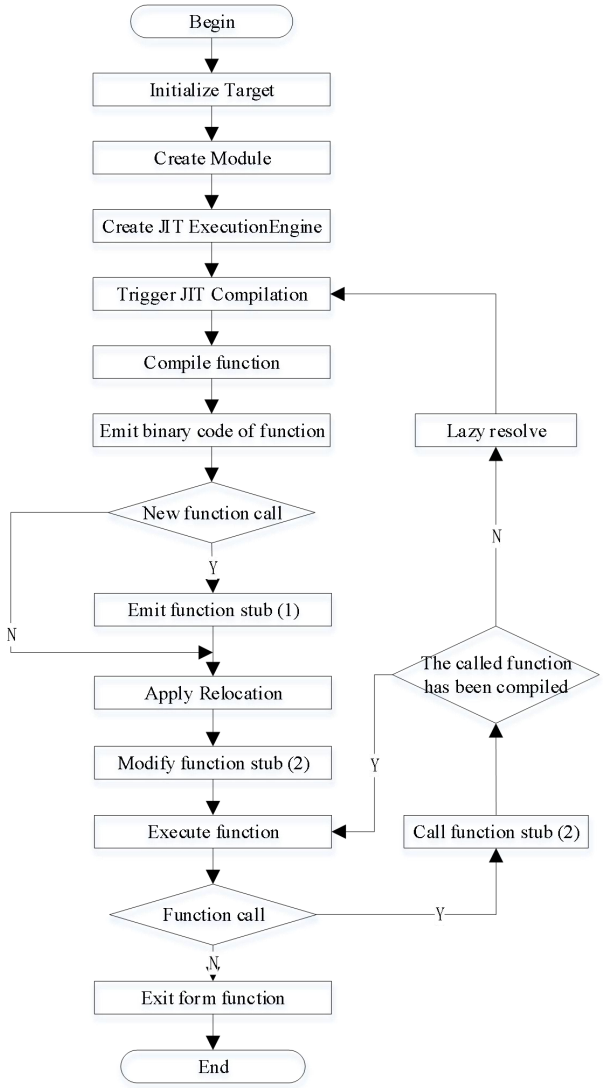

Figure 2: JIT lazy compilation implementation.

JIT lazy compilation, first initialize the target, the JIT engine will generate code for the initialized target. Then create a module that contains the LLVM IR. The IR can be obtained by parsing an existing LLVM assembly file or LLVM bitcode file, or it can be created directly in memory by calling the LLVM function creation interface. Then use the engine builder to create the JIT execution engine. When it is created, the engine builder will pass the previously created module to the JIT execution engine, and the JIT constructor will instantiate the major functional modules. Subsequently, through the JIT compilation of the function triggered by getPointerToFunction, the code generator converts the function from the LLVM IR form to the MachineInstr instruction, and then the binary code generated by the target code emitter is transmitted to the memory to complete the assembly process. If there is a function call in the function and the called function has not been compiled, a stub that jumps to the lazy parsing function is emitted through the target's JIT support interface. After the function is emitted, the relocation in the function is resolved through the target's JIT support interface to complete the linking process. At this point, the function is ready to be executed. Finally, the JIT execution engine executes the function via runFunction. If other functions are called in this function, they will enter the corresponding function stub. If the function stub has not been modified (that is, the called function is not compiled), it will enter the lazy parsing

function, compile the called function and modify the function stub with the called function address, and then enter the called function for execution; if the function stub has been modified (The called function has been compiled), JIT execution engine don't need to compile the called function again, but jump directly to the called function through the stub. After the called function finishes executing, it returns to the calling function and continues to execute the remaining instructions of the calling function until the calling function exits, and the JIT execution engine executes the exit procedure.

The JIT framework also supports eager compilation. The difference between eager compilation and lazy compilation is that when there is a function call, the called function is compiled directly instead of emitting a function stub. Before executing the function, all the functions used in the module have been compiled. And linking, make normal function calls when executing functions without lazy parsing via function stubs.

3. JIT Porting on LLVM Backend

The basic SW64 backend porting has been completed before the JIT port, which can generate complete assembly instructions for the SW64 backend. This paper does not repeat the basic backend porting content here, but focuses on the JIT-related porting and implementation details.

In the LLVM architecture, a new backend (SW64) implements the JIT feature, which can reuse existing public function modules, and only needs to add interfaces related to the new backend. Figure 3 shows the overall hierarchical structure of the JIT framework. The part above the dotted line is the target-independent function module and the porting interface provided to the backend, and the part below the dotted line is the interface that the SW64 backend needs to implement. For the SW64 backend, it is necessary to implement three main parts: a SW64 target machine that supports JIT (SW64TargetMachine), a SW64 code emitter in the code generator (SW64CodeEmitter), and a JIT support interface (SW64JITInfo).

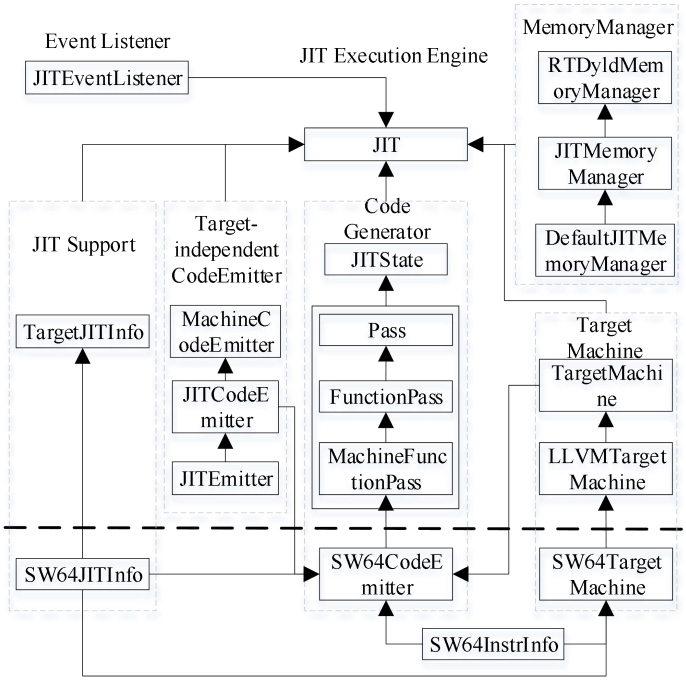

Figure 3: JIT porting hierarchy of LLVM backend.

3.1.Perfecting SW64 Target Machine

An LLVM backend with JIT support needs to achieve at least the target machine shown in Figure 4. It mainly consists of data layout, instruction description (including register description), stack and frame layout, instruction selection (including instruction matching and instruction legalization), and target JIT Support, code transmission and other parts. These components not only describe the basic properties of the target, but also perform target-related processing at various stages of code generation.

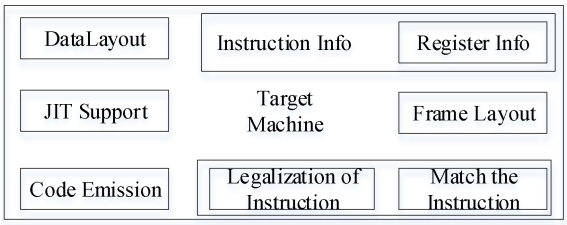

Figure 4: Structure of the target machine with JIT support.

In order to make SW64 backend support JIT, the SW64 target machine description was first perfected. The SW64 target machine description is defined in the SW64TargetMachine interface. SW64TargetMachine provides an interface to get each component, and creates a target-related configurator and configures a custom Pass in the code generator. This paper adds and improves JIT support for the SW64 target machine, and adds a pass for target code emission to the SW64 code generator, using the target code emitter to complete the assembly and linking of the code.

3.2.Implementing SW64 Code Emitter

JIT compilation does not generate assembly text. The code generator converts LLVM IR to MachineInstr, which is a very low-level intermediate representation. In the JIT framework, binary code is generated directly through the object code emitter. This paper implements the code emitter SW64CodeEmitter for the SW64 target. By creating the SW64CodeEmitter machine function pass, it encodes the MachineInstr instruction of the function and writes it to memory using the target-independent code emission interface. Figure 5 shows the relationship between the SW64 code emitter and the JIT framework.

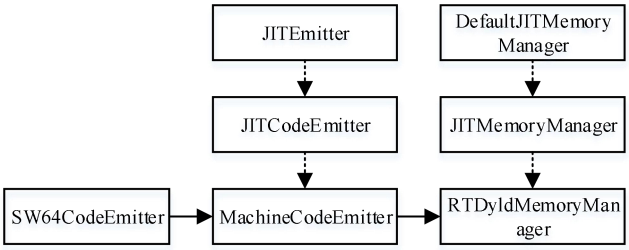

Figure 5: Relationship between SW64 code emitter and JIT framework.

First, create a SW64CodeEmitter interface that extends the MachineFunctionPass base class. The runOnMachineFunction method is used as the top-level entry point of the SW64 code emitter. This method needs to be rewritten to implement the assembly and linking process in the compilation process. The designed algorithm is: This method first allocates memory for the function and traverses all the basic blocks (MBB) in the function (F), then records the position of the basic block and encodes each valid MachineInstr (MI) in the basic block, that is, Complete the assembly

process; finally emit the auxiliary structure of the function and resolve the relocation in the function, that is, complete the linking process. Among them, the tasks of allocating memory for functions, recording the location of basic blocks, and emitting auxiliary structures are accomplished through the JITCodeEmitter interface, and the relocation of analytical functions is accomplished through the JIT support interface of SW64.

Secondly, implement the emitInstruction method to complete the coding and emission of MachineInstr. This method first checks the validity of MachineInstr, skips specific labels, and then encodes only regular instructions. In order to reduce the workload of encoding instructions, the llvm-tblgen tool automatically generates the getBinaryCodeForInstr method according to the instruction set description of the SW64 backend. This method generates binary machine code from the input MI. To obtain the encoding of each operand, this method calls getMachineOpValue by default, or calls the custom encoding method specified in the instruction description. Finally use JITCodeEmitter to emit the encoding into memory.

To encode the operands, the getMachineOpValue method is implemented to get the value of each operand. Operands have only three formats: symbol, immediate, and register. If the operand is a symbolic operand (global address, symbol, constant pool index, jump table index, basic block), use the JITCodeEmitter interface to add relocation information corresponding to the symbolic operand, and getMachineOpValue returns 0. If the operand is in immediate format, the value of the operand is returned directly. If the operand is in register format, the code of the corresponding register is returned.

Finally, in order to complete the above functions, some auxiliary functions are also written to complete tasks such as the identification of operand relocation types, the encoding of registers, and the addition of relocation entries.

3.3.Implementing SW64 JIT Support Interface

In the JIT framework, each backend needs to implement a specific JIT support interface to have JIT functions. To realize the JIT function, this paper has completed the JIT support interface of the SW64 backend. SW64JITInfo is a subclass of TargetJITInfo. The JIT execution engine accesses the SW64 backend JIT implementation through this interface. This paper completes key JIT functions such as application relocation, emission function stubs, and lazy parsing for the SW64 JIT support interface.

After the binary code of the function was emitted by the SW64 code emitter, the JIT engine could not execute the code because the linking process had not been completed. To this end, a method for applying relocations to a emitted function is implemented, using this method to iterate through all relocations in a given function, fixing each symbol reference in the currently issued function, especially relocations using the GOT table To point to the correct memory address. At present, the analysis of 7 relocation types for SW64 targets has been implemented.

In JIT lazy compilation, since normal function calls are replaced with stub calls, the JIT support interface is required to emit stub code that jumps to the called function to delay compilation of the called function. This paper implements the method of emitting function stubs for the JIT support interface of the SW64 target. This method first determines the target address of the function stub: the address of the compiled called function or the address of the lazy parsing function. The SW64 target provides custom function stub layout information with a size of 24 bytes and 4-byte alignment (including 6 instructions). JITCodeEmitter uses this stub information to allocate space for it before issuing a new stub. Because when the JIT framework was established, there were no dependable interfaces for tasks that issued independent instructions outside the basic block, so the instructions of function stub need to be written manually: the target address is encoded into 6

instructions to achieve the jump to the target function turn. Finally, use the JITCodeEmitter object to emit the function stub code into memory, and set this stub code to have executable permissions.

In JIT lazy compilation, the function stub emitted by the JIT support interface will jump to the lazy parsing function when the called function is not compiled. This paper writes this function for the SW64 JIT support interface. This function is written in assembly code and has complete control over register save and restore. First write the assembly code to open a certain stack space and save the value of the register (reservation register, parameter register, etc.) in the stack space, then write the code that calls the callback compilation function with the address of the function stub as a formal parameter, and then write the code to restore the register, and finally write the jump code to make the function exit into the stub modified by the callback compilation function, and then execute the called function.

This paper implements the compilation function for the SW64 JIT support interface. The callback compilation function of the JIT support interface plays a vital role in JIT lazy compilation. Finally, this method triggers JIT compilation of the called function. This method first resolves the stub of the called function to the address of the called function through the stub parsing function of the JIT engine. The called function has not been compiled. At this time, JIT compilation will be triggered, and the compiled called function address will be obtained. Then use the address of the called function to rewrite the function stub. The process of rewriting the function stub is similar to the process of transmitting the function stub. When the called function is called again, the calling function jumps directly to the called function through the function stub instead of calling the lazy parsing function through the function stub.

In addition to the above functions, other JIT-related functions are implemented for the SW64 JIT support interface, such as: obtaining a lazy parser function to initialize the parser instance of the JIT code emitter, providing a lazy parser function for the SW64 target for the parser, and The SW64 target provides a stub parsing function to implement the interaction between the JIT execution engine and the target in lazy compilation.

4. Experiment and Analysis

4.1. Test Method

LLVM provides an interpreter tool (lli) to use the JIT engine. lli implements the LLVM bitcode interpreter and JIT compiler by using the LLVM execution engine. This paper uses the LLVM official test-suite to verify the correctness of the ported JIT compiler.

The Test-suite contains the complete source code of the programs, which are written in C or C++ and can be compiled and linked into an executable file. The JIT test is different from the usual static compilation test. The process is shown in Figure 6. When testing, the Test-suite tested program is compiled using the clang compiler and a set of flags to generate LLVM assembly code. Then use the optimizer opt to optimize the intermediate code to generate optimized LLVM bitcode. lli then runs the bitcode with a set of options and dumps the output to a file. Finally, the output is compared with the reference output to check the correctness of the program output.

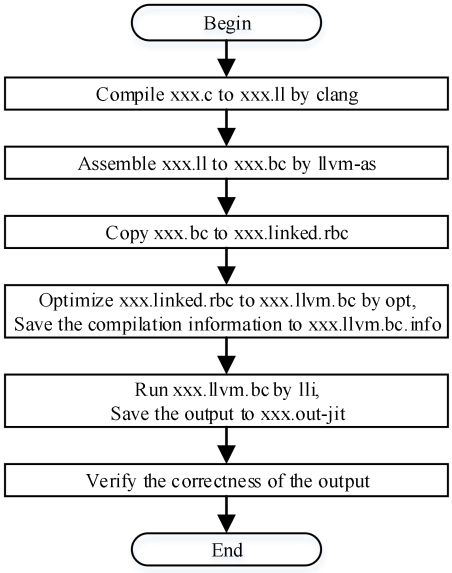

Figure 6: JIT test flow chart.

To eliminate the impact of LLVM itself, this paper set the Intel platform with X86_64 architecture as a control group for verification. Compile the unmodified LLVM3.3 source code on the Intel platform and then test it; the source code of modified and support the Sunway processor to compile and test on the Sunway platform.

After the porting, the correctness of the JIT compiler is verified on Sunway server. The CPU is SW1621, a 64-bit processor with frequency of 1.60GHz, with deepin Linux operating system.

X86_64 architecture control group: The CPU is Intel (R) Xeon (R) CPU E5-2682 v4, with frequency of 2.50GHz, the operating system is CentOS7.

4.2.Test Results

After testing, the overall results are shown in Table 1.

Table 1: Test result of test-suite.

Platform	Test	Pass	Fail	Pass Ratio
Sunway	483	471	12	97.52%
Intel	483	471	12	97.52%

4.3.Results Analysis

On the whole, the pass rate of test-suite on Sunway platform reached 97.52%, which is consistent with the Intel platform, and has the same error cases and number, which is enough to prove the correctness of the porting work and verify the reliability of the transplanted JIT compiler.

The test does not include cases that are only supported by the X86_64 architecture. Twelve cases that failed the test, the first ten are cases related to exception handling. They reported errors during runtime, and the two platforms were consistent. The output of the case scimark2 is consistent between the two platforms. The problem with case tls is that the current porting version does not yet support Thread Local Storage, and the X86_64 backend JIT does not support Thread Local Storage ether.

5. Conclusions

This paper first briefly introduces the LLVM JIT framework, including its main components and the functions of each part, and analyzes the implementation principle of JIT lazy compilation. Then analyzed the backend porting mechanism of JIT and completed the SW64 backend porting.

The porting work of this paper makes LLVM JIT compile support Sunway processor, and passed 97.52% cases of test-suite, reaching the level of Intel platform, verifying the correctness of the transplanted JIT compiler. At present, the SW64 backend has not yet implemented support for Thread Local Storage and does not support case of recompilation. As the work continues, while improving the above functions, the quality of the generated code can be further optimized and improved to further improve the efficiency of JIT compilation.

References

[1] http://llvm.org/.

[2] Chris Lattner. The architecture of open source applications: LLVM [Online]. Available: http://www.aosabook.org/en/llvm.html.

[3] Lopes, Bruno Cardoso. Getting Started with LLVM Core Libraries. Packt Publishing, 2014.

[4] Abel Avram. Apple Speeds Up WebKit's JS Engine with LLVM JIT[Online]. Available: https://www.infoq.com/news/2014/05/safari-webkit-javascript-llvm/, 2014-5-16.

[5] Michael Larabel. PostgreSQL Begins Landing LLVM JIT Support For Faster Performance [Online]. Available: https://www.phoronix.com/scan.php?page=news_item&px=PostgreSQL-LLVM-JIT-Landing, 2018-3-22.

[6] Yu-Hsin Tsai, I-Wei Wu, I-Chun Liu, Jean Jyh-Jiun Shann. "Improving performance of JNA by using LLVM JIT compiler." Computer and Information Science (ICIS), 2013 IEEE/ACIS 12th International Conference on IEEE, 2013.

[7] Michal Gregor, Juraj Spalek. "Using LLVM-based JIT Compilation in Genetic Programming." Elektro. IEEE, 2016.