Redundant Register Saving Elimination for Procedure Calls

A Link-time Optimization using Cross-procedure Register Usage Analysis

CHIH-YUNG LIANG, National Taiwan University
SHIH-KAI LIN, National Taiwan University

Student ID 1 R05944012
Student ID 2 R05922043
Repository https://github.com/ntu-homeworks/compiler-project

1 INTRODUCTION

Instruction Set Architectures (ISA) like *X86* have limited number of general-purposed registers. Although it is more desired to store many kinds of values in registers than in the stack, *register spills* happen whenever the amount of free registers becomes insufficient. Especially, lots of registers are spilled on function call instructions to follow the calling convention. As what calling conventions define, if a register is categorized as preserved, it is expected to retain the original value after a function call. That is, the callee has to save/restore such registers in the function entry/exiting point if they are used. On the other hand, if a register is non-preserved, it is not guaranteed to keep the original value after a function call. The caller is responsible to save the register before the call.

To reduce the number of spills, several complex algorithms performing global (inter-basic-block) register allocation are proposed and implemented on modern compilers. However, the register allocation is usually limited within the scope of a function because of the tremendous increased complexity and the separation compilation. Although sometimes the amount of register is actually sufficient to not spill, redundant save/store instructions are still inserted into functions to follow the calling convention.

This project, as a link-time optimization, utilizes cross-procedure register usage analysis to eliminate register saves and restores. Briefly, the strategy is that a callee-saved (preserved) register used in a function is no longer saved if it is sure to be originally dead (that is, the value in that register can be discarded) in all possible callers. In advance, the optimizer also tries to rename a register to another one fulfilling that strategy to eliminate the saving. Built on LLVM, the project is specific to the X86 backend to prove the concept.

2 STRATEGIES

There are many strategies decided during the development of this project to achieve the best effect, including the calling convention, function processing order, register liveness detection, and register reassignment. This section describes these strategies in detail.

2.1 Calling Convention

To simplify the implementation of this project, a calling convention for preserved register set is defined for the code compiled by this project:

$$\begin{cases} \text{Preserved (callee-saved): } rbx, rbp, r10, r11, r12, r13, r14, r15 \\ \text{Non-preserved (caller-saved): } rax, rcx, rdx, rsi, rdi, r8, r9 \end{cases}$$

This calling convention minimizes the non-preserved register set to as much as possible prevent callers from saving registers. It results in that registers are mostly saved by the callee during each

function call, so we can focus on eliminating register saving in callee. In detail, when processing a function, the optimizer analyzes the register status in all the function's callers and determines whether to remove some register saves.

2.2 Function Processing Order

When compiling an LLVM module (a source file), LLVM emits each function one by one. To be more specific, until a function is completely emitted, the next function is not processed, including the register allocation. Since the optimization depends on register allocation results of all callers, it is desired that all callers are already processed before processing the callee. As a result, functions in a module are reordered to the depth-first traversing sequence of the call graph of that module.

2.3 Register Liveness Detection

A register is dead if the value stored in it is no long used. If a callee-saved register is found dead in all possible callers, saving that register becomes redundant and can be discarded. However, it is not necessary that a function can always find the register allocation information of all its callers. It must match all these conditions:

(1) The function does not have external linkage. Even this optimization is performed in link-time, a function with external linkage is still possible called by unknown callers.
(2) The function does not have address taken. The function may be called indirectly through function pointers.
(3) All callers have their register allocation done.

In fact, as the optimizer reorders functions in the processed module (mentioned in 2.2), condition (3) is very likely to fulfill. On the other hand, there is an optional experimental exception that all registers are dead before entering the function "*run*", although this function does not satisfy all three conditions.

There are two approaches implemented in this project to find out whether a specified register is dead in a specified caller. Both approaches may have uncertain answer to the liveness of the register, but either one of these approaches can confirm that the register is dead. In addition, if the register is confirmed dead in all callers, it is a dead register that does not require saving. However, to ensure the correctness, the optimizer has to be conservative that a register is seen as alive if both approaches make uncertain answers.

2.3.1 Linear Liveness Searching. This approach searches instructions in the basic block the caller lives. Starting from the call instruction, it performs a backward search followed by a forward search on number of instructions to realize the register liveness. The searches are stopped if there comes a certain answer. The backward search can tell a register is

alive if the register is just defined or read, and the definition is not dead, or
dead if the register is killed, clobbered, or has a dead definition.

On the other hand, the forward search can tell a register is

alive if the register is read, or
dead if the register is clobbered or fully redefined.

If there is still no a certain answer after performing these searches, this approach cannot reply an effective answer.

2.3.2 Dead Register Table. This approach maintains a table recording dead register sets of each function. The optimizer can realize if a register must be dead in some callers by looking up this table. To build the table, for each function the optimizer processes, it retrieves the dead register sets

of all callers by looking up this table and using the approach mentioned in 2.3.1. The intersection of these sets is the initial dead register set of the function. Then, the initial set is subtracted with all registers the function touches to produce the final result set, which contains registers that must be dead in the whole function. Finally, the table is updated with the information of the dead register set of this function. Notice that due to the exception of the function "*run*", the initial dead register set of "*run*" contains all registers on the machine.

2.4 Register Reassignment

Saves and restores of a register in a function cannot be discarded if the register is not sured to be dead in all possible callers of the function. However, through register renaming, the optimizer may find that the newly assigned register is known to be dead in all possible callers. Thus, the optimizer will try to rename each register whose saves and stores cannot be eliminated, including its sub-registers. To rename a register, besides to make sure the new assignment is dead in all callers (in order to be profitable), there are several constraints to meet to make the renaming legal:

- The new assignment is not reserved (e.g. *RSP*, *RIP*, and sometimes *RBP*).
- The new assignment will not be clobbered during a function call. That is, either it is a preserved (callee-saved) register or the function does not contains any function call instruction.
- The new assignment is originally free in the function. That is, neither itself nor its sub-registers is referenced in the function.
- The new assignment can be referenced by instructions in the function in the same way the original assignment does. For example, in case the register *RBX* is accessed by some instructions through *AH*, it cannot be renamed to *R12*, which cannot be addressed like that.

3 IMPLEMENTATION DETAIL

This project directly modifies the source code of LLVM, and there are two additional LLVM passes inserted into the process LLVM emits machine/assembly code for a module. The first pass, *TopDownFunctionReorderPass*, is a module pass, which performs Intermediate Representation (IR) transformation, and is inserted into the code generation preparation stage. The other pass, *X86CallSpillEliPass*, is a machine function pass, which performs machine IR transformation, and is inserted into the pre-emit stage.

3.1 TopDownFunctionReorderPass

This pass reorders functions in the processed module according to the depth-first traversal sequence of the call graph to achieve the strategy mentioned in 2.2.

3.2 X86CallSpillEliPass

There is two jobs this pass does: (1) To build the dead register table, which is mentioned in 2.3.2. (2) To eliminate register saves and restores in function entry and exiting points. The first job is always performed at the last stage of the pass, even the second job has failed. All the implementation of this pass follow strategies mentioned in Section 2. However, there are still some technical problems to issue.

3.2.1 Caller Finding. The machine instruction calling the function being processed is actually difficult to find as there aren't one-on-one mappings from IR instructions to machine instructions. However, after modifying *MachineFunctionAnalysis*, IR functions and machine functions are one-on-one mapped, so the actually caller machine instruction can now be found by searching from the machine function mapped from the IR function the caller IR instruction lies in.

3.2.2 Register Saves/Restores Instruction Finding. Register save instructions are always in the function entry point, and restore instructions are always in the function exiting points. However, while the entry point of the function is always the first basic block of the function, the exiting points of the function are relatively difficult to find. Moreover, there may be multiple exiting points. To find exiting blocks of the function, the optimizer can only search for possibly-return blocks between all basic blocks in the function.

4 USAGE

This section describes how to build and test the project.

4.1 Building

```
$ git clone --depth 1 https://github.com/ntu-homeworks/compiler-project.git
$ mkdir build && cd build
$ cmake -DCMAKE_BUILD_TYPE=Release -DLLVM_ENABLE_ASSERTIONS=On \
        -DLLVM_TARGETS_TO_BUILD=X86 ../compiler-project
$ make llc
```

4.2 Testing *llc*

For the fairness, please test the baseline using the same *llc* to apply same calling convention to compare the result! Also, mixing code with different calling convention (e.g. calling a library function like *printf*) can cause error!

```
llc [-x86-call-spill-eli \
                [-eli-call-spill-depth=<Liveness Check Depth>] \
                [-no-livein-run=<true|false>] \
                [-debug -debug-only=x86-eliminate-call-spill] \
                [-stats] \
    ]

-x86-call-spill-eli          Turn on the optimizer. Default is off.
-eli-call-spill-depth=       The number of instructions to search in the method
                             mentioned in 2.3.1. Default is 10.
-no-livein-run=<true|false>  To use the experimental exception for function
                             "run" mentioned in 2.3. Default is on.
-debug                       To dump debug information.
-stats                       To dump statistical results.
```

5 EVALUATION

We only apply our optimizer on the module being tested, where the module contains a procedure "*run*" as the entry point. This module is compiled into LLVM IR using *Clang 3.9* with flags including `-emit-llvm`. Then, the module is optimized in IR level with *opt 3.9* flagged by `-O3`. After that, the machine code of this module can be emitted by our modified *llc 3.9*. Finally, the emitted module is linked with the wrapper used to test that module. Since the wrapper is compiled with unmodified compiler, it may have different calling convention with the tested module. Thus, there is a calling convention protector included in the wrapper. On the other hand, to evaluate the optimization result, besides statistical numbers dumped by our optimizer, we use *Pin Tool* to count instructions executed in each function of the tested module.

Table 1. Example Output of Test Cases

Test Case	Output
Simple	-553,279,039
Recursive	1,484,417,395
Indirect	-587,190,255
ExternalCall	-483,017,440

Table 2. Statistical Result of the Optimizer

	Result in Emitted Code			Measured in Runtime		
Test Case	Eliminated Spills	Renamed Registers	Removed Instrs.	Original Instr. Count	Optimized Instr. Count	Reduced Instr. Count
Simple	7	4	14	4,800,012	3,800,008	20.8%
Recursive	4	1	8	2,100,042	1,700,038	19%
Indirect	4	1	8	4,300,013	3,900,009	9.3%
ExternalCall	0	0	0	40,000,012	40,000,012	0%

5.1 Test Cases

We have tested the project with four test cases, which are either used to test the correctness ot used to test the ability to eliminate redundant spills:

For elimination ability (1) *Simple* (the example test case).
For correctness (1) *Recursive*, (2) *Indirect*, and (3) *ExternalCall*.

The test case *Recursive* test if the optimizer can correctly handle recursive function call; that is, the register spills in such functions cannot be eliminated. The test case *Indirect* tests if the optimizer accidentally eliminates spills in functions being called indirectly. The test case *ExternalCall* tests if the optimizer accidentally eliminates spills in exported functions. Table 1 shows the supposed output of each test case.

5.2 Statistical Results

Table 2 demonstrates our optimizer result in (1) number of register spills eliminated, (2) number of registers renamed in callee, and (3) number of instructions removed. We also show the instruction count measured in runtime for each test case before and after the optimization. Columns about *Instr. Count* in Table 2 are the total numbers of instructions executed by the function *run* and all its children, recursively. Notice that the data about the original instruction count is also under the calling convention we apply, which is mentioned in 2.1.